A study of fault tolerance techniques for associative processors*

by BEHROOZ PARHAMI and ALGIRDAS AVIŽIENIS

University of California
Los Angeles, California

INTRODUCTION—ASSOCIATIVE PROCESSING

Associative processing techniques have been suggested for numerous application areas and have been proven to be superior to more conventional procedures for a number of specialized applications.[1] Recent advances in computer technology and development of new architectural concepts for associative devices have made the design of larger and more flexible systems possible. Such systems are extremely complex and must be adequately protected against failures. This paper reports on the results of a study[2] which has indicated the techniques that are applicable and difficulties that may be encountered in the design of fault-tolerant associative processors.

In the remainder of this section, we will briefly review the four basic organizations for associative processors; i.e., fully parallel, bit-serial, word-serial, and block-oriented. This discussion is motivated by the fact that each of these organizations requires a different treatment for some fault tolerance considerations, such as the detection of failures. This classification is based on the degree of parallelism in operations or, alternatively, the amount of storage associated with each unit of processing logic. A more detailed discussion of these concepts and a comprehensive set of references can be found in Reference 1.

In fully parallel associative processors, processing logic is associated with each bit of stored data. Most fully parallel systems implement only the exact-match search operation in hardware and use software techniques for arithmetic, logic, and more complex searches. An associative processor has been proposed[3] in which a variety of comparison and arithmetic operations are performed in parallel on each word.

In bit-serial associative processors, processing logic is associated with each word of stored data. All the words can be processed in parallel, each in a bit-serial manner. Bit-serial systems represent a compromise between fully parallel and word-serial systems and can be economically implemented with state-of-the-art technology[4] since they can utilize conventional storage elements.

In word-serial associative processors, a single processing unit operates serially on all the words.[5] This approach essentially represents hardware implementation of a simple program loop which is used for linear search. The elimination of instruction fetching and decoding time and the high data rates that can be achieved by circulating memories contribute to the relative efficiency of this approach as compared to programmed linear search.

In block-oriented associative processors, one block of information is associated with a unit of processing logic. A low-cost implementation of such a system may use a head-per-track magnetic recording memory in which each block is stored on one or more tracks.[6] Block-oriented organization is particularly suitable for applications such as information storage and retrieval where a large storage capacity is required.

FAULT TOLERANCE OF ASSOCIATIVE PROCESSORS

Based on the applications that have been proposed for associative devices, there are at least three reasons for studying the fault-tolerance problems of such devices: (1) In some proposed application areas for associative processors, such as air traffic control,[7] the effect of an undetected fault-induced error may be catastrophic; (2) To be able to perform control functions[8] in a fault-tolerant computer, an associative device must itself be fault tolerant, since, otherwise, it will become part of the system's hard core and will contribute heavily to its unreliability; (3) The extreme complexity of large, general-purpose associative processors necessitates the incorporation of fault tolerance features into their design.

It is remarkable, therefore, that the problem of fault-tolerance of associative devices has remained virtually untouched. Ewing and Davies[4] give techniques for coping with some hardware malfunctions in a plated-wire implementation of a particular associative processor. Furthermore, they are only concerned with detecting such errors and disabling the corresponding cell. Fault detection is done by performing certain operations periodically. Proudman[9] suggests that a single error correcting code can be used in conjunction with mismatch detectors with a threshold of 2. However, this scheme is not valid if logic or masked write operations have to be performed, since such operations destroy the coding. Lipovski[10] presents an associative processor architecture in

* This research was supported by the National Science Foundation, Grant No. GJ 33007X.

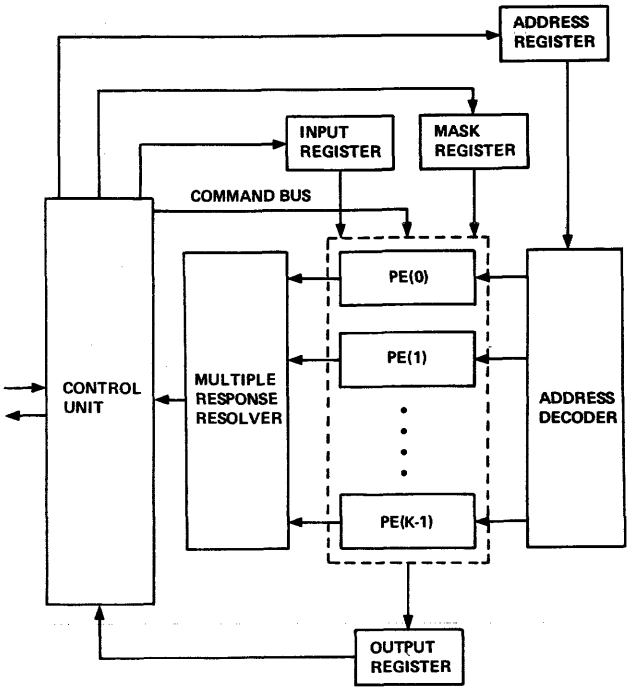


Figure 1—General model for an associative processor

which the processing elements are connected into a tree structure. He contends that such a system is fail-soft since faulty subtrees can be easily isolated from the rest of the system. However, he does not indicate how faults are detected.

In the remainder of this paper, we will identify and discuss some techniques that are applicable in the design of fault-tolerant associative processors. We will concern ourselves with hardware faults and will assume the programs to be correct representations of intended algorithms for the specified domain of operation. We may note, however, that the simplified software of associative processors (e.g., fewer loops), with respect to conventional systems, results in a proportional simplification in the problem of software fault tolerance. A summary of the results presented here has been published elsewhere.[11]

Figure 1 shows a model for an associative processor which applies to all of the classes described in Section 1 except for word-serial systems. Since word-serial associative processors closely resemble conventional systems, their fault tolerance problems can be studied separately. Each processing element (PE) in Figure 1, consists on one unit of processing logic and its associated storage elements. In general, the processing elements in the PE array communicate with each other and the exact pattern of intercommunication is application-dependent.

A study of fault-induced errors in an associative processor shows that they are not easily detectable since a single fault may cause an arbitrary number of errors. This is evident for faults in global subsystems of Figure 1, such as the input and mask registers. For example, in a search operation a smaller, larger, or an entirely different set may respond.[2] A single fault in one processing element may cause errors in others because of PE intercommunication, making concurrent fault detection highly desirable. The problem is further compounded by the fact that each PE performs logic and selective write operations on individual data bits which as we know are not easily checkable without a high level of redundancy.[12]

The selection of applicable redundancy techniques is the most important step in the design of fault-tolerant digital systems. The first basic choice is between static (masking) and dynamic (replacement) schemes. The advantages of dynamic redundancy schemes over static ones are well-known.[13] For associative processors, there are at least two other advantages to the dynamic redundancy approach: (1) The high degree of internal complexity makes the implementation of a statically redundant associative processor very costly and inconvenient; (2) The highly regular structure of a major part of an associative processor (PE array) lends itself naturally to modularization. Such modules can be made identical in structure and can share spare modules.

Let us assume that the associative processor of Figure 1 is divided into M modules, each consisting of P processing elements. Figure 2 shows a possible structure for each module if the decoding and multiple response resolution functions are distributed among the modules. As shown in Figure 2, the information regarding the responses is passed serially through the modules. Clearly fully parallel and mixed series-parallel schemes can be used in much the same way as carry-

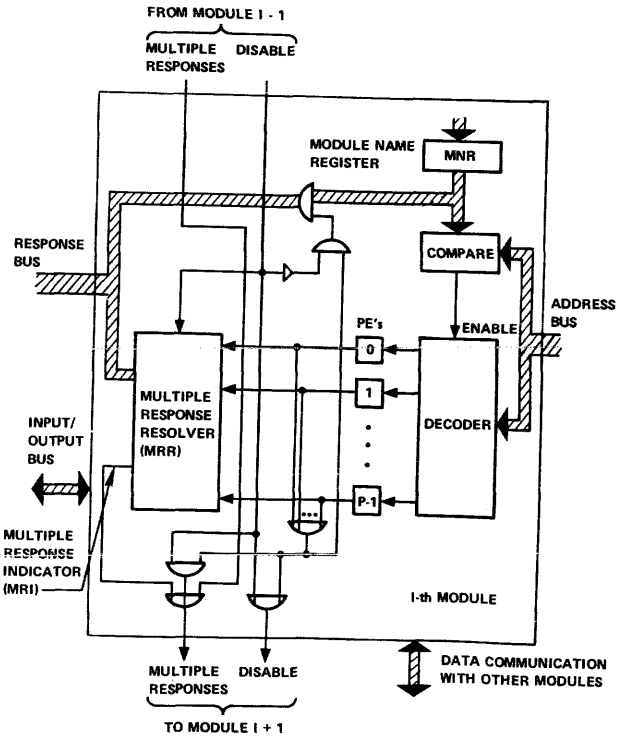


Figure 2—Organization of a module in a modular associative processor

lookahead circuits for adders. Figure 3 shows the modules and their interconnections. One-dimensional intercommunication between modules will be assumed for simplicity.

Given a modular associative device as shown in Figures 2 and 3, it can be made fault tolerant by the following steps: (1) Incorporating internal failure detection ability within each module; (2) Adding S spare modules; and (3) Designing switching mechanisms and corresponding algorithms for reconfiguration. We will assume that the M+S operating and spare modules are permanently connected to the main data buses and that special isolating circuits exist between each module and the data buses. Therefore, reconfiguration takes place by "power switching" and by providing alternate intercommunication paths between modules.

ERROR DETECTION TECHNIQUES

As noted earlier, the problem of error detection in associative processors is a difficult one and conventional coding techniques are generally not applicable. However, there are special cases where low-redundancy coding techniques can be used. We now discuss some such special cases with respect to the four classes of associative processors mentioned earlier. This discussion will be followed by a brief introduction to the self-checking design technique which is applicable in all cases.

A fully parallel associative memory with only "exact-match" search operation and without masking capability can be protected by using a code with a minimum distance of d. With this scheme, if conventional mismatch detectors are used, stored words containing $d-1$ or fewer errors will never respond to a search operation (there is always at least one mismatch signal) and are effectively isolated from the rest of the system until periodic diagnosis routines detect their failure. On the other hand, if mismatch detectors with a threshold of $d \div 2$ are used, up to $k = \lceil d \div 2 \rceil - 1$ bit errors can be masked by the search logic; i.e., a word containing k or fewer errors will still match its original value (k or fewer mismatch signals) and will not match any other value ($k+1$ or more mismatch signals). The difficulty is that such an

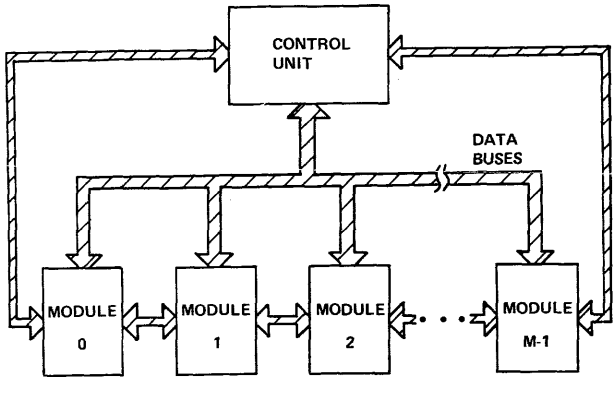


Figure 3—Module intercommunication in a modular associative processor

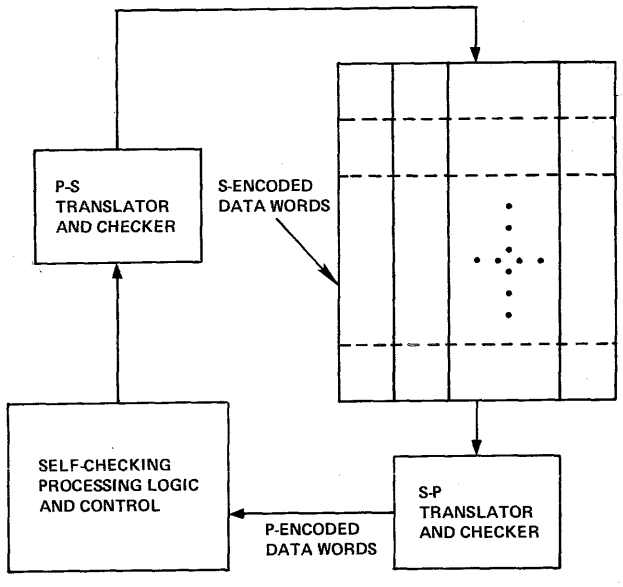
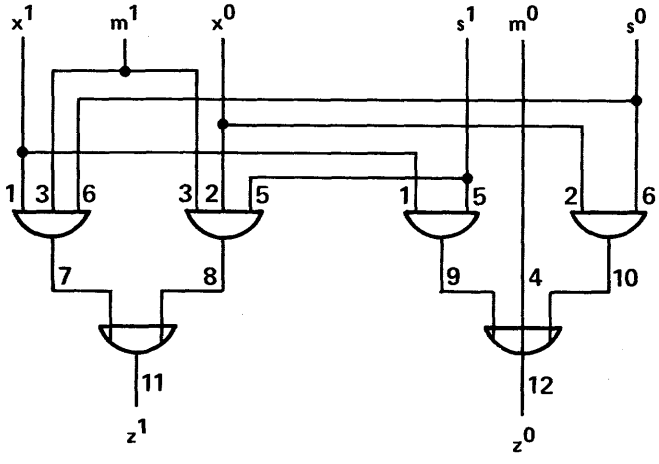


Figure 4—A fault-tolerant word-serial associative processor

associative device will have no application besides simple table look-up. For most other applications, masking capability and more complex search types are essential. Also, in associative processors, arithmetic and logic operations need to be performed. Clearly, low-redundancy codes are not applicable for such operations.

Considerations for bit-serial systems are similar to those for fully parallel systems. One advantage which exists here is the serial processing of bits in each word. This allows us to artificially extend each operation to the entire word by performing "null" operation on bit positions not originally specified. Now, since all the bits of each word are processed serially, codes with low-cost serial encoding and decoding can be used to protect against storage errors. Simple parity checking is particularly attractive because of the small amount of additional circuitry required for encoding and checking. It should be noted, however, that if operations on small fields within the words are to be performed frequently, the above scheme may result in a significant reduction in speed. Also, operations on multiple fields within the same word (e.g., adding two fields and storing the sum in a third field) do not lend themselves to this approach unless a complete circulation is used for each bit operation, resulting in an almost intolerable speed reduction.

As noted earlier, because processing is performed serially in a word-serial system, protection against failures becomes relatively simple. Low-redundancy coding can be used to protect against storage errors. Failures in the processing logic may be detected through self-checking[14] design. Self-checking translators may be needed to convert the storage encoding (S-encoding) to an encoding suitable for processing (P-encoding). The main requirement on the P and S encodings is that fast (parallel) translation between the two must be possible. This is true since the data rates achieved by circulating memory devices are very high (bit rates of



INPUT PATTERNS						TESTABILITY OF LINES S-A-1, S-A-0, OR NONE										
															1	1	1
1	2	3	4	5	6	1	2	3	4	5	6	7	8	9	0	1	2
0	1	0	1	0	1							1	1			1	0
0	1	0	1	1	0			1	0			1	1			1	0
0	1	1	0	0	1	1	0			1	0	1	1		0	1	0
0	1	1	0	1	0	1	0	0	1	0	1			0	1	1	0	1
1	0	0	1	0	1			1	0			1	1			1	0
1	0	0	1	1	0							1	1			1	0
1	0	1	0	0	1	0	1	0	1	1	0	0		1	1	0	1
1	0	1	0	1	0	0	1			0	1	1	1	0		1	0

Figure 5—A two-level realization of two-rail masked comparison and its testability with code-space inputs

10-100 MHz). Figure 4 shows a possible configuration for a fault-tolerant, word-serial associative processor. Since during each operation cycle, the entire memory content is circulated through the processing logic, 2-dimensional codes may be used for additional protection against storage errors, if desired.

One favorable property of block-oriented systems with respect to fault tolerance is that during each operation cycle a processing element operates on the entire block of information assigned to it. This enables the use of block codes which result in relatively low redundancy and have simple serial checking algorithms. The simplest possible scheme is to use a parity bit per block of information which detects all single errors. However, if mechanical storage devices are used, error bursts become very probable due to dust particles, minute scratches, or defects in the oxide coating. It has been noted that low-cost arithmetic error codes are very effective for coping with such burst errors.[15] The checking algorithm for these codes is very simple and requires little additional hardware if an adder is already present in each PE.[6]

As can be seen from the previous discussion, low-redundancy coding techniques are applicable only in special cases. Design of logic circuits in self-checking and self-testing form[14] (i.e., in a way that internal circuit failures manifest themselves on the circuit's output and such that each failure is detected by the circuit's normal inputs) particularly if 1-out-of-2 encoding is used appears to be promising. However, because of the relatively higher complexity of the self-checking design approach as compared to low-redundancy coding techniques, this approach should be used when others fail or for protecting the system's hard core.

A detailed discussion of self-checking design concepts is beyond the scope of this paper. Instead, we present as an example a self-checking circuit for masked comparison of two bits. Denoting the mask bit by m, data bit by x and the stored bit by s, the mismatch result z is defined as

$$z = m \cdot (x \oplus s);$$

i.e., we have a mismatch if the given bit position is not masked ($m=1$) and the data bit x does not equal (match) the stored bit s. Figure 5 shows a two-level, self-checking. and self-testing realization with two-rail encoding of the variables; i.e., a variable y is represented by a pair (y^1, y^0) with $y^1 = y$ and $y^0 = \bar{y}$ during error-free operation. It is easy to show that any single-line failure results in the correct output or one of the "illegal" combinations (0,0) or (1,1) on the output. Hence, the circuit is self-checking. The fact that the circuit is self-testing is verified by applying an APL/360[16] program called TESTDETECT[17] to it. The table given in Figure 5 is the output of the TESTDETECT program applied to a description of the given circuit. An entry of 1(0) in this table indicates that the corresponding input pattern detects s-a-1 (s-a-0) failure of the given line by producing on the circuit's output one of the "illegal" combinations (0,0) or (1,1). Figure 5 indicates that each line in the given circuit is tested for both s-a-1 and s-a-0 failures during normal operation.

RECONFIGURATION TECHNIQUES

For a modular associative device to tolerate module failures, the module interconnections should not be rigid as shown in Figure 3. Rather, the modules should be interconnected through specially designed switching circuits which prevent a system failure as a result of the failure of a module. The setting of these switching mechanisms determines the

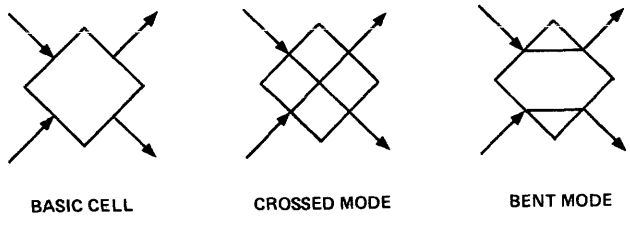


Figure 6—A two-state switching cell

system configuration and can be changed by a central monitor if required. If a module error is indicated and the existence of a permanent failure is determined, reconfiguration procedures must be initiated to establish a new working configuration. In general, data transfers between modules and correction of fault-induced errors is needed as part of the reconfiguration process.

As will be seen the additional complexity introduced by the modularization overhead and reconfiguration switching mechanism is an increasing function of the total number of modules M+S. Therefore, improving the reliability by increasing M and S is possible only to a certain point. Therefore, the optimal module size, in terms of the number of processing elements it contains, and the number of spare modules must be determined for each application through tradeoff studies involving reliability improvement and the corresponding increase in cost.

In the remainder of this section, we will assume only unidirectional (left to right) data flow between the modules in Figure 3. The generalization of the results to bidirectional data exchange is straightforward. After detecting the existence of a faulty module, the following steps must be taken before normal operation can resume: (1) Locating the faulty module; (2) Determining a new working configuration; (3) Initiating appropriate data transfers; and (4) Effecting reconfiguration through switching. The criteria that should be used in evaluating each scheme include: (1) The amount of data transfers needed; (2) The complexity of the reconfiguration algorithms; (3) The number of spares S needed for tolerating f module failures; and (4) The complexity of additional switching circuitry.

A straightforward solution is the use of a "permutation network"[18] which can interconnect the modules in any order. Such a permutation network can be implemented as a cellular array[19] of two-state basic modules shown in Figure 6. Since the complexity of such a cellular permutation network is roughly proportional to the square of the number of modules, its use can be justified only if a relatively small number of modules are involved.

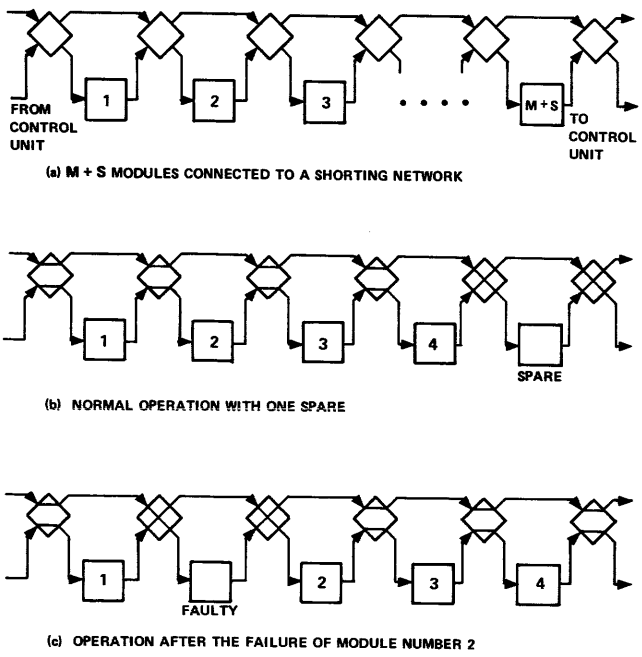


Figure 7—Reconfiguration with a shorting network

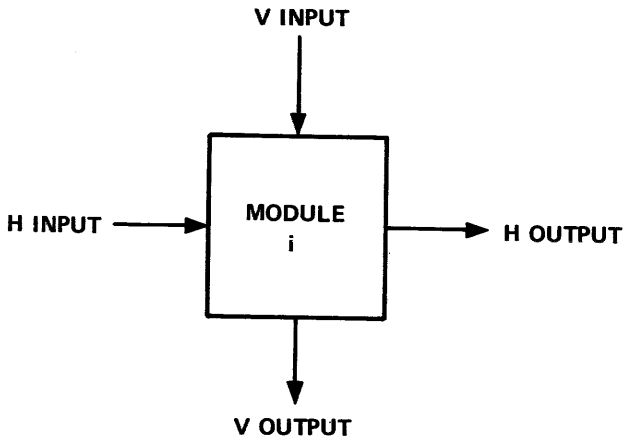


Figure 8—Basic module for distributed reconfiguration

The basic module of Figure 6 can be used in a different way to form a "shorting network."[18] As shown in Figure 7, such a shorting network can be used to route data around the faulty and spare modules. One disadvantage of these schemes, particularly as shown in Figure 7, is the excessive amount of data transfers needed in the case of a failure. The number of transfers needed can be reduced by optimal placement of the spare modules. It can be shown[2] that data transfers are minimized if the kth spare module is in position

$$i_k = k + (k - 0.5) \times \mathrm{M} \div \mathrm{S}$$

for $k = 1, 2, \ldots, \mathrm{S}$. For example, with $\mathrm{M} = 6$ and $\mathrm{S} = 3$ and the modules numbered 1, 2, . . . , 9, the spares should be in positions 2, 5, and 8. Intuitively, this corresponds to dividing the string of M+S modules into S roughly equal groups and placing a spare in the middle of each group. Reference 2 also contains APL/360 algorithms for and examples of reconfiguration with shorting networks.

Another approach to the reconfiguration problem is the use of a distributed switching mechanism; i.e., distributing the switching function among the modules. This can be done by providing each module with a *set* of input and output lines instead of one as shown in Figure 3. Then, if a successor module connected to one module output fails, a module connected to another output can act as its successor. The simplest case, which will be discussed here, is when each module has two sets of inputs and two sets of outputs. As shown in Figure 8, the two inputs and two outputs are distinguished by the letters H and V (horizontal and vertical). The module has four states denoted by HH, HV, VH, and VV, depending on whether the H or V input is used and whether the output is generated on the H or V output.

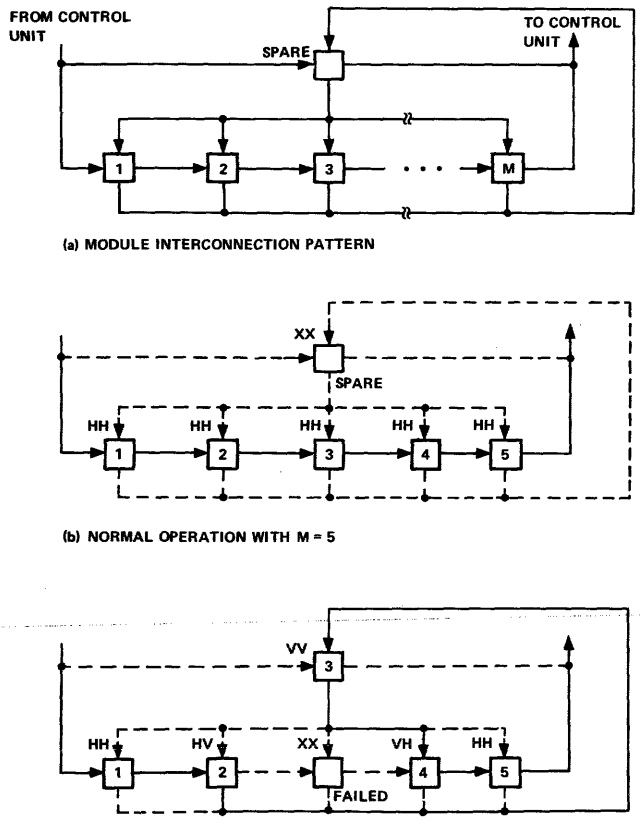


Figure 9—Distributed reconfiguration with simple sparing

Figure 9 shows how modules of Figure 8 can be interconnected in a simple sparing scheme with S=1. The spare module can replace any one of the operating modules and only one module's data need to be transferred in the event of a failure. For S>1, this scheme can be used if the M operating modules are divided into S groups each having one spare. The disadvantages of this scheme are: (1) only one module failure can be tolerated in each group; (2) If a faulty module is not reliably powered off, it may produce meaningless data on the common connection to the spare module.

Figure 10 shows a two-dimensional arrangement of the basic modules. It can be seen in Figure 10 that all 9 modules can be connected into a string similar to Figure 3 by appropriate selection of module states. If any single module fails, the remaining 8 can continue their operation. Double module failures will leave at least 6 usable modules. Hence, with M=8 and S=1, this scheme can tolerate all single module failures. With M=6 and S=3, all double failures can also be tolerated as well as some triple failures. Note that if both successors of a module fail, it cannot be used. Hence the tolerance of two module failures requires three spare modules. Two interesting and equivalent unsolved problems exist for the two-dimensional arrangement of modules: (1) Given M+S modules with M required to be operating, how should one interconnect them to tolerate the maximum number f of failures? (2) Given the requirement for M operating modules and tolerance of f failures, what is the minimum number S of spares required and the corresponding interconnection pattern?

The basic advantage of this scheme is that the switching mechanism is not part of the system's hard core since a failure in the switching circuits is equivalent to a module failure. The working configuration is supported solely by the non-failed modules. The only place where interference from failed modules may result is on the output bus. This can be avoided by using an output selector circuit to isolate the modules from the bus. The main disadvantages of this scheme are the complexity of the reconfiguration algorithm, excessive data transfers, and tolerance of fewer than S failures. APL/360 algorithms have been written for the reconfiguration process.[2]

It is interesting to note that in a rectangular two-dimensional configuration (Figure 10) with r rows and c columns, one can obtain bounds on the number of modules in various states. Let us denote by n_{HH} the number of modules which

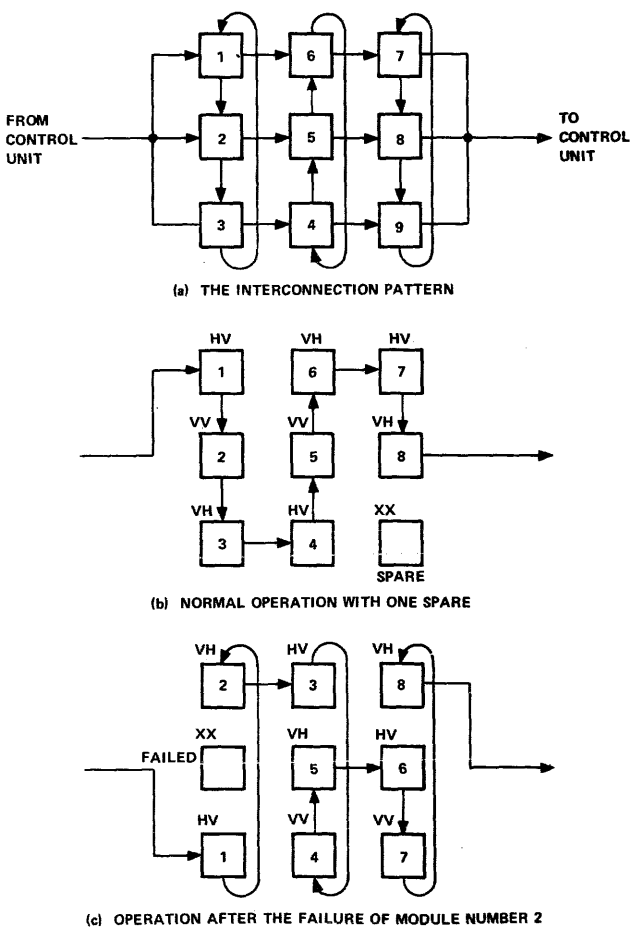


Figure 10—A two-dimensional arrangement of basic modules

are in state HH. Similarly, define n_{HV}, n_{VH}, and n_{VV}. It can be shown that if M is the number of operating modules, then:[2]

$$n_{HH} \leq (r \times c - M) \div (r-1)(M-c) \div (r-1)$$
$$\leq n_{HV} = n_{VH} \leq c\,M - 2c \times$$
$$\leq n_{VV}$$
$$\leq (M-c) \times (r-2) \div (r-1)$$

Such bounds are useful in verifying the correctness of a given configuration.

A FAULT-TOLERANT ASSOCIATIVE PROCESSOR

In this section, we illustrate the applicability of some of the techniques discussed previously by presenting the design and evaluation of a fault-tolerant associative processor called SPARE (inverse acronym for Error-tolerant and Reconfigurable Associative Processor with Self-repair). SPARE is essentially a fault-tolerant version of an associative processor which has been described previously.[4] Figure 11 shows a block diagram of the non-redundant system. The random-access memory is used for storing instructions and constants and consists of 4096 24-bit words. The associative memory contains 512 96-bit words. External data can be transferred directly to either one of the memories under automatic interrupt control.

The non-redundant associative processor of Figure 11 can be divided into two parts: (1) The associative (parallel) section, which consists of the associative memory array, bit column selection logic, and word logic; (2) The control and

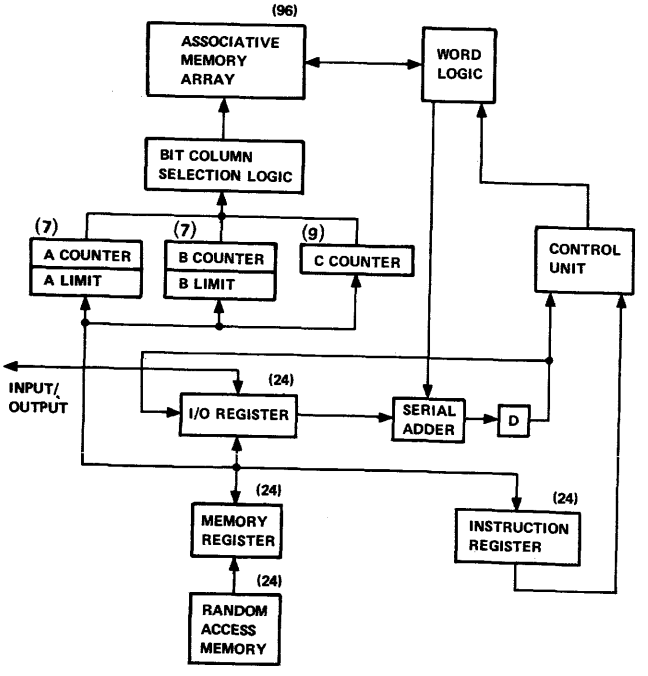


Figure 11—Block diagram of the non-redundant associative processor

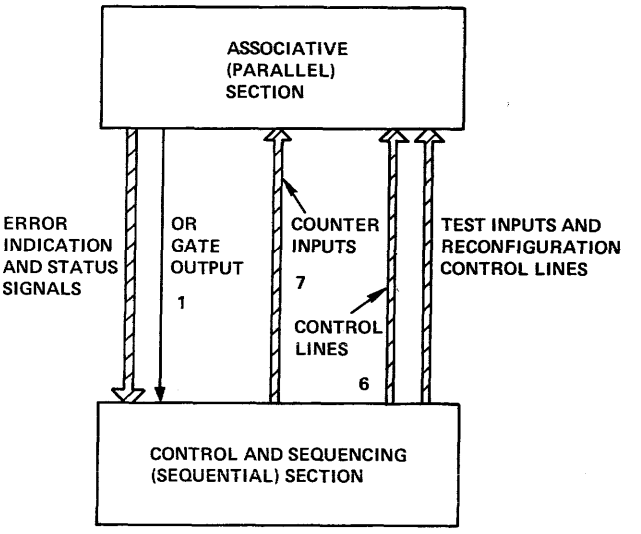


Figure 12—The parallel and sequential sections of SPARE and their interface

sequencing (sequential) section, which contains all other subsystems of Figure 11. Figure 12 shows the parallel and sequential sections of SPARE and their interface requirements. The sequential section uses the status signals and test inputs for monitoring the operation of the parallel section. We now briefly discuss the three main features of SPARE; i.e., error tolerance, reconfigurability, and self-repair.

To achieve *error tolerance*, the parallel section of SPARE is divided into M indentical modules. S spare modules are shared by the operating modules. Each module has internal failure detection capability which is provided by self-checking design of its circuitry using two-rail encoding of logic variables. When a module error is indicated to the sequential section, the recovery mode is entered and the final result may be the replacement of the faulty module by a spare module. The sequential section of SPARE resembles a small general-purpose computer and can, therefore, be made fault tolerant by conventional techniques.

One of the very important properties of associative processors is simple modular growth. The size of an associative processor can grow without a need to alter its algorithms. This suggests that if additional processing capability is required, the redundant processing logic in SPARE can be utilized. Even the two channels of the two-rail circuits can be used independently to double the processing capability if certain design criteria are met.[2] Specifically, we postulate the following operation strategy for SPARE: (1) During normal operation the system works in redundant mode with a number of spare modules; (2) If a module failure occurs or additional processing capability is needed and if a sufficient number of spares are available, they are switched in; (3) If a module failure occurs or additional processing capability is needed and spare modules are not available, the system

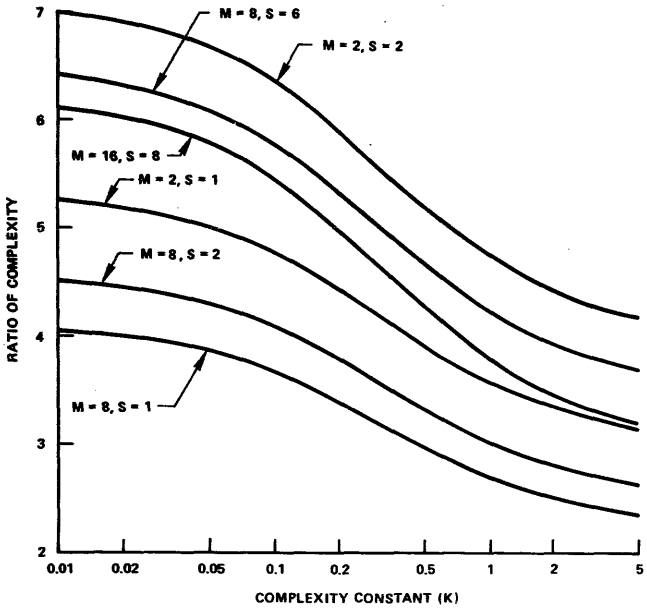


Figure 13—Relative complexity of SPARE as a function of K

reconfigures into simplex mode by utilizing the two channels of the two-rail circuits independently.

Of the reconfiguration techniques discussed earlier, the one using a permutation network seems to be suitable for SPARE since only one intercommunication line (two in self-checking design) exists between modules and the number of modules is expected to be small (M = 4 or 8, for example). The *self-repair* process will then essentially consist of computing and setting of a new state for the permutation network. This process must be followed by a recovery procedure to transfer the data stored in the failed module to the one which replaces it. The permutation network has a two-rail self-checking design but no spare is provided for it.

The detailed design of SPARE[2] shows that if K is the relative complexity of one storage bit with respect to a logic gate, the hardware complexity (cost) of various designs, in terms of gate equivalents, are as follows:

Non-redundant system	NRC = 15872 + 49152 × K
Permutation network	PNC = −31 + 25 × (M + S)²
Each self-checking module	MC = 512 × (110 + 192 × K) ÷ M
Redundant system	RC = PNC + (M + S) × MC

The value of K is technology-dependent and has been chosen as a parameter for generality. Figure 13 shows the ratio of complexity for the redundant and non-redundant designs as a function of K for several configurations of SPARE. In computing the complexity factors, we have only considered the parallel section of SPARE and have ignored the sequential part.

To compute the reliability of SPARE, we will assume that the coverage factor C includes the reliability of the permutation network. Using the reliability modeling technique of Bouricius et al.,[20] we find the following reliability equations directly

$$R_{nr}(T) = \exp(-\lambda_{nr} \times T)$$

$$R_r(T) = \exp(-M \times \lambda_m \times T) \sum_{i=0}^{S} \binom{M+i-1}{i} \times \{C[1-\exp(-\lambda_m \times T)]\}^i$$

where T is the mission time, λ_{nr} and λ_m denote the failure rates for the non-redundant system and a self-checking module, and R_{nr} and R_r denote the non-redundant and redundant reliabilities, respectively. Figure 14 shows the reliability improvement factor defined as $[1-R_{nr}(T)] \div [1-R_r(T)]$ as a function of mission time T for several configurations of SPARE.

From the preceding discussions we conclude that for mission times which are short compared to the MTBF for the non-redundant system, a significant increase in reliability is possible with a relatively small number of spare modules. A more detailed study of the effect of mission time T, coverage factor C, and complexity constant K on the optimal configuration of SPARE is being carried out. (For a given set of values for T, C, and K, an *optimal configuration* is defined

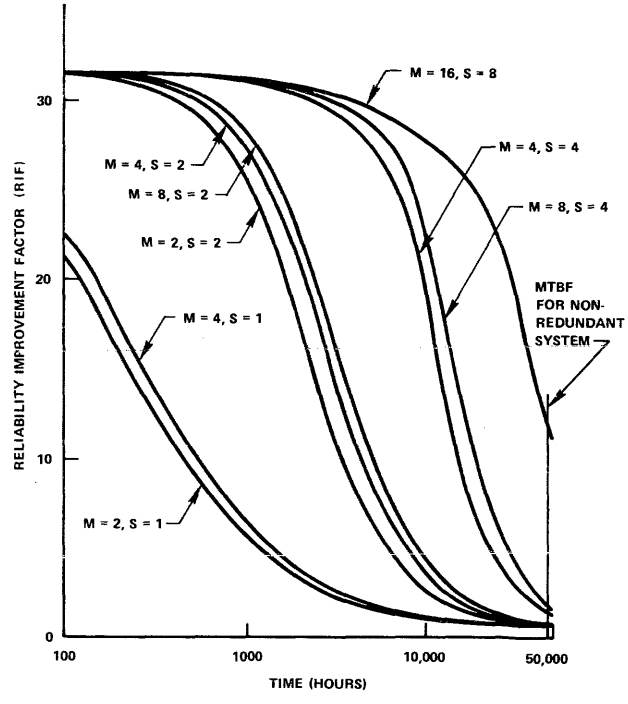


Figure 14—Reliability improvement for various configurations of SPARE with respect to the non-redundant system (K = 0.1, C = 0.99)

as a pair of values for M and S which result in a lower system cost than all other pairs with the same or higher reliabilities.)

CONCLUSION

In this paper, we have presented the results of a study on the fault-tolerance of associative processors. Our main conclusions are as follows:

(1) Dynamic redundancy is to be preferred over static approach because associative processors lend themselves naturally to modularization and since spares can be shared by a number of identical modules;
(2) Low-redundancy coding techniques are applicable for error detection in associative processors but only in special cases. In particular, the use of arithmetic error codes for block-oriented systems appears to be promising;
(3) Application of self-checking circuit design techniques seems to be an attractive alternative for error detection in associative devices;
(4) Complex switching mechanisms and algorithms need to be devised to enable the sharing of spares by a collection of identical modules which communicate with each other.

Further research is needed in two equally important areas. The first area is the design of completely checked digital circuits. Systematic techniques need to be developed to aid the designers in choosing suitable input and output encodings and producing a self-checking design when presented with a non-redundant circuit or its functional behavior. Cost and effectiveness studies are also needed for the self-checking design approach to determine the increase in complexity over non-redundant designs and the actual error detection coverage which it provides.

The second area is general techniques for reconfiguration in array processors. Extension and generalization of the results presented here are possible in two directions. First, one can conceive of other interconnection schemes for the case where one-dimensional intercommunication exists between modules. For example, we may consider a three-dimensional interconnection pattern in which there are three choices for each of the left and right neighbors for a module. Second, one may seek generalizations to the cases where multi-dimensional module intercommunication is used. This is a considerably more complex problem. As an example, it may be possible to embed a two-dimensional intercommunication pattern into a three-dimensional or four-dimensional matrix of interconnected modules. Then, each module can choose its left, right, upper, and lower neighbors in the same manner as a module could select its left and right neighbors in the case of one-dimensional intercommunication.

Also, we have not considered the testing aspects of associative processors. This is an important area for future investigation since the design of a fault-tolerant associative processor must be initially verified through testing. In addition, for an associative processor which is dedicated to a certain task, there is frequently some idle time which can be utilized by periodic diagnostic routines.

REFERENCES

1. Parhami, B., "Associative Memories and Processors: An Overview and Selected Bibliography," *Proceedings of the IEEE*, Vol. 61, No. 6, pp. 722-730, June 1973.
2. Parhami, B., "Design Techniques for Associative Memories and Processors," Technical Report UCLA-ENG-7321, Computer Science Department, University of California, Los Angeles, March 1972. (Also published as a Ph.D. dissertation.)
3. Shore, J. E. and F. A. Polkinghorn, *A General-Purpose Associative Processor*, Naval Research Lab. Report, Washington, D.C., March 1969.
4. Ewing, R. G. and P. M. Davies, "An Associative Processor," *AFIPS Conference Proceedings*, Vol. 26 (1964 Fall Joint Computer Conference), Spartan Books, Baltimore, Maryland, 1964, pp. 147-158.
5. Crofut, W. A. and M. R. Sottile, "Design Techniques of a Delay Line Content-Addressed Memory," *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 4, pp. 529-534, August 1966.
6. Parhami, B., "A Highly Parallel Computing System for Information Retrieval," *AFIPS Conference Proceedings*, Vol. 41 (1972 Fall Joint Computer Conference), AFIPS Press, Montvale, New Jersey, 1972, pp. 681-690.
7. Thurber, K. J., "An Associative Processor for Air Traffic Control," *AFIPS Conference Proceedings*, Vol. 38 (1971 Spring Joint Computer Conference), AFIPS Press, Montvale, New Jersey, 1971, pp. 49-59.
8. Berg, R. O. and M. D. Johnson, "An Associative Memory for Executive Control Functions in an Advanced Avionics Computer System," *Proceedings of IEEE International Computer Group Conference*, June 1970, pp. 336-342.
9. Proudman, A., "Bulk Associative Memory with Error Correction," *IBM Technical Disclosure Bulletin*, Vol. 12, No. 7, pp. 1076-1077, December 1969.
10. Lipovski, G. J., "The Architecture of a Large Associative Processor," *AFIPS Conference Proceedings*, Vol. 36 (1970 Spring Joint Computer Conference), AFIPS Press, Montvale, New Jersey, 1970, pp. 385-396.
11. Parhami, B. and A. Avižienis, "Design of Fault-Tolerant Associative Processors," *Proceedings of the First Annual Symposium on Computer Architecture*, Gainesville, Florida, December 1973, pp. 141-145.
12. Peterson, W. W., and M. O. Rabin, "On Codes for Checking Logical Operations," *IBM Journal of Research and Development*, Vol. 3, No. 2, pp. 163-168, April 1959.
13. Avižienis, A., "Design of Fault-Tolerant Computers," *AFIPS Conference Proceedings*, Vol. 31, (1967 Fall Joint Computer Conference), Thompson Books, Washington, D. C., 1967, pp. 733-743.
14. Carter, W. C. and P. R. Schneider, "Design of Dynamically Checked Computers," *Information Processing 68*, (Proceedings of IFIP Congress, Edinburgh, Scotland, August 1968), North Holland Publishing Company, Amsterdam, 1969, pp. 878-883.
15. Parhami, B. and A. Avižienis, "Application of Arithmetic Error Codes for Checking of Mass Memories," *Digest of International Symposium on Fault-Tolerant Computing*, Palo Alto, California, June 1973, pp. 47-51.
16. Falkoff, A. D. and K. E. Iverson, *APL/360 User's Manual*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, August 1968.
17. Bouricius, W. G., W. C. Carter, K. A. Duke, J. P. Roth and P. R. Schneider, "Interactive Design of Self-Testing Circuitry," *Proceedings of Purdue Centennial Year Symposium on Information Processing*, Lafayette, Indiana, April 1969, pp. 73-80.

18. Levitt, K. N., M. W. Green, and J. Goldberg, "A Study of Data Commutation Problems in a Self-Repairable Multiprocessor," *AFIPS Conference Proceedings*, Vol. 32 (1968 Spring Joint Computer Conference) Thompson Book Company, Washington, D.C., 1968, pp. 515-527.
19. Kautz, W. H., K. N. Levitt, and A. Waksman, "Cellular Interconnection Arrays," *IEEE Transactions on Computers*, Vol. C-17, No. 5, pp. 443-451, May 1968.
20. Bouricius, W. G., W. C. Carter, and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," *Proceedings of the 24th National Conference of ACM*, San Francisco, California, August 1969, pp. 295-309.