A Fault-Tolerant Architecture for an Automatic Vision-Guided Vehicle

MANSUR R. KABUKA, MEMBER, IEEE, SURJADI HARJADI STUDENT MEMBER, IEEE, AND AKMAL YOUNIS

Abstract—**A fault-tolerant architecture for an automatic navigation system is presented. The system employs a mixed type of architecture in which the speed advantages of both pipelined and parallel architectures are exploited to achieve real-time navigation. The fault-tolerant architecture is presented using two reconfiguration strategies. To evaluate the proposed architecture, its reliability, availability, and safety are investigated using Markov models. In addition, the feasibility of implementing the proposed architecture is studied.**

I. Introduction

THE study of automatic guided vehicles (AGV) has received a great deal of attention in the past decade due in part to the increasingly complex modern transportation problem and to the vast area of related applications that involve monotonous and tedious tasks or hazardous environments.

In fact, AGV's can be considered one of the key factors in flexible manufacturing systems. These systems are used to maximize the throughput by attempting to equalize the workload among their various components. This can be achieved by premeditated task planning and devising of efficient navigation methods for the autonomous transporters. These methods should be flexible, inexpensive, and easily modifiable. Initial efforts in obtaining these methods for robot navigation included the use of buried wires [1] and painted lines [2], [3]. The buried-wire method is still the most popular among the Japanese manufacturers of mobile robots [4].

Research has concentrated lately on navigation with little or no *a priori* knowledge of the surrounding environment. Since the knowledge about the environment is minimal, the system must depend on its sensing mechanisms for navigation. Most of the current research has relied upon visual sensors [5]–[13], ultrasonic rangefinders [14], tactile sensors [15]–[18], and laser rangefinders [19]–[21]. These systems are usually designed with the aim of being completely autonomous. However, most mobile robots are still inept in this regard, the limitations defined finally by their sensing capabilities. Realistic implementation of completely autonomous robots capable of navigating any unknown environment has yet to be achieved, although great advancements have been made. On the contrary, if the environment is controllable—a likely situation in indoor industrial settings—the situation is simplified and practical solutions could be implemented.

In a controlled environment, the AGV usually navigates by correlating the information previously stored about the environment with the information it gathers along its way. One way of providing this information is in the use of marks and patterns that can be discretely placed within the environment. The AGV determines its position relative to these marks and subsequently locates itself in the environment. Some of the marks already developed include laser-detected corner cubes [19] and retroreflective "spot marks" [22].

Artificial intelligence also has been used for purposes of path planning and navigation [23], [24]. If the environment is assumed to be known completely, algorithmic methods can be employed to plan the path as a one-time off-line operation [25]–[27]. Although this can prove to be fast for path-planning execution, it does not account for the possibility of the path being blocked due to temporary reasons. In such a situation, the AGV's should have the ability to revise, during navigation, their previously planned paths so as to obtain optimal or suboptimal solutions [28]–[30].

To increase the efficiency and minimize the chance of accidents, an AGV system must be designed with a maximum regard towards reliability, availability, and safety. In this paper, a fault-tolerant architecture is proposed for the implementation of the AGV presented in [29] and [30] to attain these properties. No matter how well designed the system is, its circuit components may fail at a crucial moment. This can prove to be hazardous not only to the AGV but also to its environment. In a less serious condition, the AGV can lose its way and wander until rescued, with the consequence of wasting both time and efficiency. Moreover, the interruption of material flow caused by malfunction can seriously disrupt the whole system's operation. Under more serious circumstances, the AGV can collide with another AGV, causing massive destruction that could have been avoided with the use of a fault-tolerant AGV. On the other hand, a fault-tolerant AGV will continue its work uninterrupted and thus inflict a positive influence on overall productivity.

Manuscript received February 24, 1988; revised August 8, 1989.

The authors are with the Department of Electrical and Computer Engineering, University of Miami, P. O. Box 248294, Coral Gables, FL 33124.

IEEE Log Number 8932019.

0018-9472/90/0300-0380$01.00

The navigation system, presented in [29] and [30], is briefly described in Section II. The fault-tolerant architecture of the system is presented in Section III. An evaluation of the proposed architecture is investigated in Section IV, in which reliability, availability, and safety are discussed using Markov models. In Section V, a feasibility study of the proposed architecture is conducted to illustrate that it could be implemented to perform the required real-time navigation of the AGV. Also, this feasibility study could be considered, in a wider context, useful for implementing any application in which the transformation of a path network map into an interconnected graph is needed.

II. AGV System Description

The AGV goes through three different phases to achieve real-time navigation.

a) *Path network learning:* During this phase a scaled map of the path network is presented to the system via a camera. The system automatically extracts information, such as location and number of incident edges of a node, node adjacency, edge path direction trace, and edge length, using image processing techniques. Information, such as width and height of an edge, whether an edge is directed or not, and the maximum load an edge can support, is input to the system interactively and stored in the path network database. The learning process is not entirely carried out in an interactive mode due to the fact that it would have been time consuming and prone to human errors.

b) *Automatic path scheduling:* In this phase, the information in the path network database is used to generate a path that could be traced by the AGV to navigate from the source node to the destination note. The requested navigation could be specified to the system in one of three possible alternatives:

   1) initial, destination nodes;
   2) initial, destination, and part of the path crossing nodes;
   3) initial, destination, and all the path crossing nodes.

c) *Real-time navigation:* During real-time navigation, the AGV extracts information about the environment and confirms it with the previously planned path. For the purpose of guidance, a finite-state machine is designed to control the navigation of the AGV. The AGV identifies nodes by decoding a bar code attached to each node and detects its current position by monitoring a painted line on the floor. If an unexpected node is reached, or a path is found blocked by an obstacle, a double heuristic search technique is used to generate a new path, to guide the system back to the required destination.

For the AGV to perform the previously mentioned tasks, it utilizes a multiple instruction multiple data stream (MIMD) architecture in which three parallel paths are identified.

1) *Node / edge pipelined unit (NPU):* This path consists of four image-processing pipelined modules used for segmentation, smoothing, thinning, and node/edge extraction. The NPU first transforms the input image of the path network into a graph of interconnected segments whose widths are one pixel in each direction. This graph is processed to extract information about network nodes and their interconnecting edges. The extracted information is stored in order to be used for automatic path planning.
2) *Bar-code pipelined unit (BPU):* When the AGV navigates along a path, it decodes the bar codes of the nodes it encounters using the BPU. The decoded information is compared with the stored information about the path in order for the AGV to determine if it is on the right path.
3) *Obstacle avoidance unit (OAU):* This unit is used for detecting unexpected obstacles using an ultrasonic rangefinder. For the occasion that an obstacle is detected, by observing both a discontinuity in the painted line and a shorter range than the expected distance to the next node, the authors developed an algorithm in which the AGV attempts to maneuver around it, if possible, or else backtracks to the last reached node and a new path is replanned. Ultrasonic obstacle avoidance has been chosen because it is inexpensive, easy to implement, and gives the required range information. The range data obtained is considered of acceptable resolution because it is used for detection, not for recognition.

III. AGV Fault-Tolerant Architecture

Since the AGV consists of three parallel paths, each of which uses a pipelined architecture as shown in Fig. 1, it is difficult to apply fault-tolerance design techniques to the system as a whole. Hence a modular approach is used in which each of the three parallel paths is designed independently.

In this paper, the issue of fault tolerance is addressed for the NPU, due to its inherent complexity as compared to the BPU and OAU, but the concept could similarly be applied to these units. Two architectures are presented for achieving fault tolerance in the NPU. Although both architectures are able to recover from two faults only, their concept of operation could easily be extended to include any number of faults m.

In the first architecture, shown in Fig. 2, a direct replacement strategy is employed to reconfigure the system after fault detection and location. In this strategy, one of the spares is downloaded with the state and program of the faulty processor in order to replace it when normal operations are resumed after recovery. While

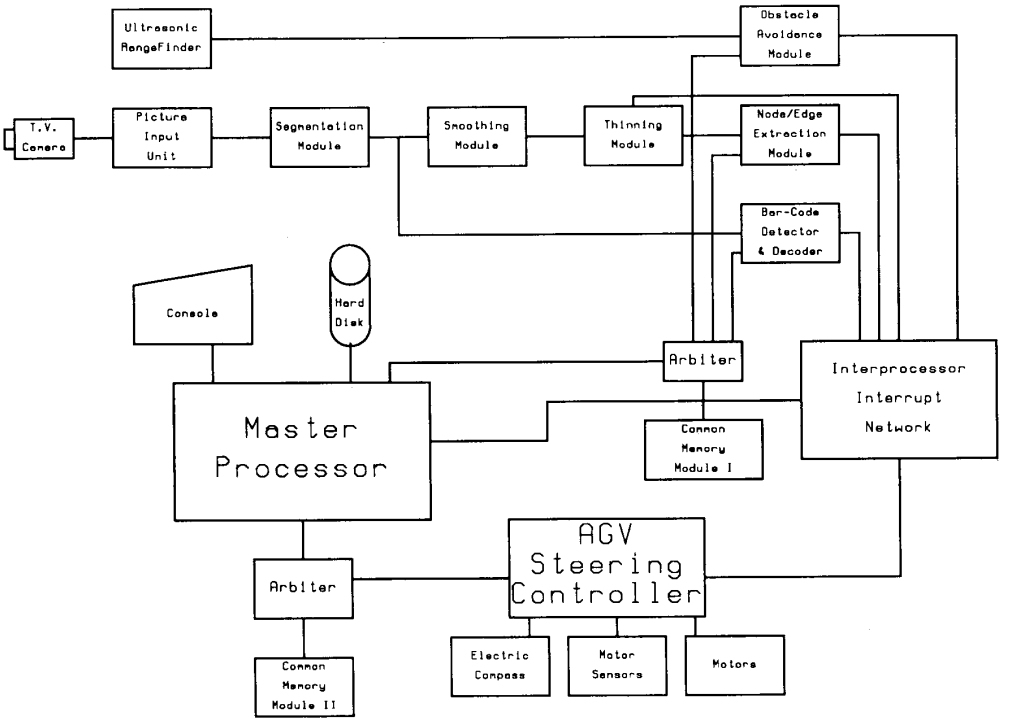


Fig. 1. AGV system architecture.

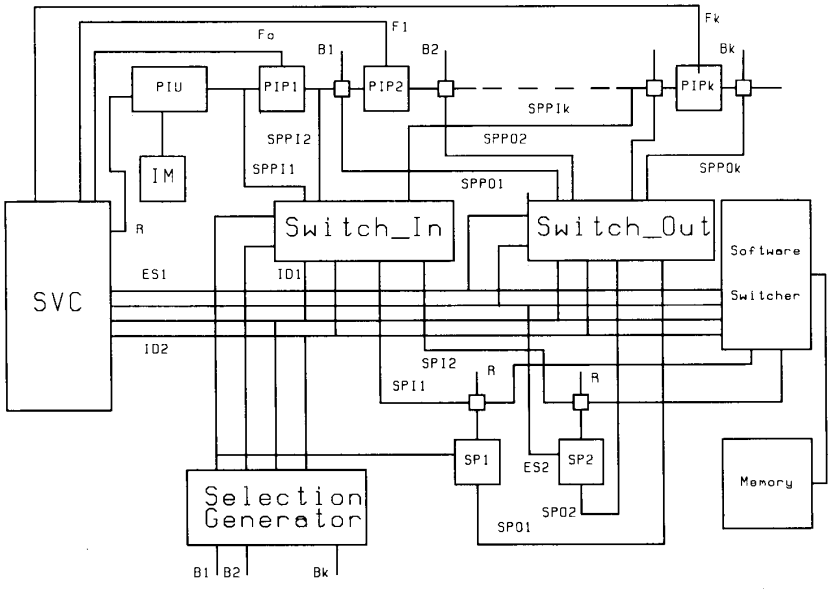


Fig. 2. Direct replacement architecture.

in the second architecture, shown in Fig. 3, a rippling replacement strategy is employed in which a faulty processor is replaced by its successor in the pipeline, and that successor is in turn replaced by its successor, and so on, until the last processor in the pipeline is replaced by one of the spare processors. Hence the reconfiguration process results in a shift-of-functions operation that starts at the faulty processor and ends up at one of the spare processors.

In both architectures, each module of the NPU is assumed to consist of a number of similar processors connected in a pipeline, with the total number of processors in all four modules equal to k. Both architectures employ the concept of standby sparing by having two

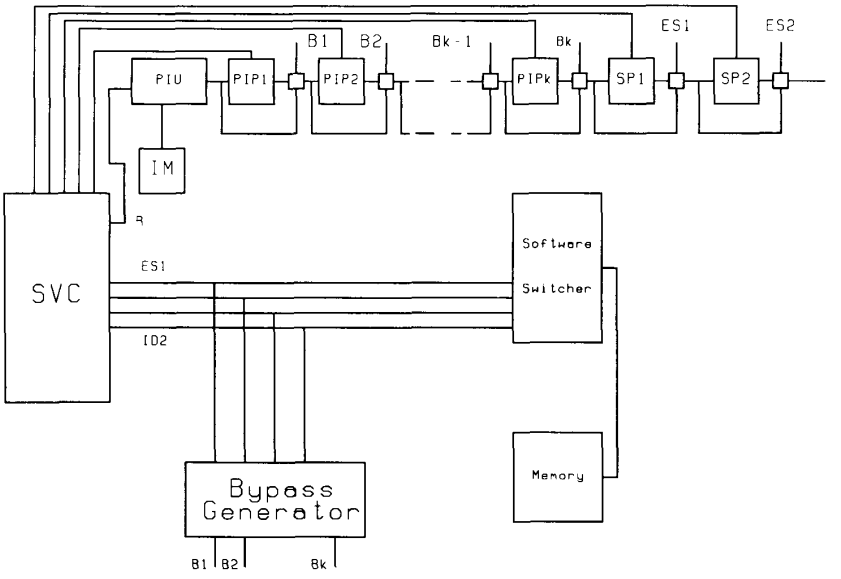


Fig. 3. Rippling replacement architecture.

spare processors ready to be substituted for faulty processors, once one or two faults have been detected and located. They differ mainly in the reconfiguration strategy, by which a faulty processor is replaced by one of the spare processors.

Moreover, to achieve their respective strategies, they use a combined hardware/software switching mechanism, which is initiated when a fault is detected. The hardware portion is responsible for establishing proper physical links between the spare processors and the nonfaulty processors, while the software portion is responsible for downloading the states and programs needed for each architecture so as to get the spare processor functioning in the pipeline.

A. Direct Replacement Architecture

The direct replacement architecture proposed for the NPU is shown in Fig. 2, where the following basic units are identified.

a) *State verification controller (SVC):* This unit is responsible for scanning and testing the pipelined processors to globally detect any faults. On the other hand, faults are detected locally within each processor running its own self-diagnostics; when a fault is detected, it alters an internal register containing its order (state) in the pipeline. When one or two faults are detected by the SVC, it generates the codes of the faulty processors on its output lines (ID_1, ID_2) and activates the corresponding enable lines (ES_1, ES_2). These signals are used in turn by the other units in the system to initiate the switching mechanism. Each of the *ID* codes has s bits, where s is given by $\lceil \log_2 k \rceil$, so that each pipelined processor is given a unique identifying code.

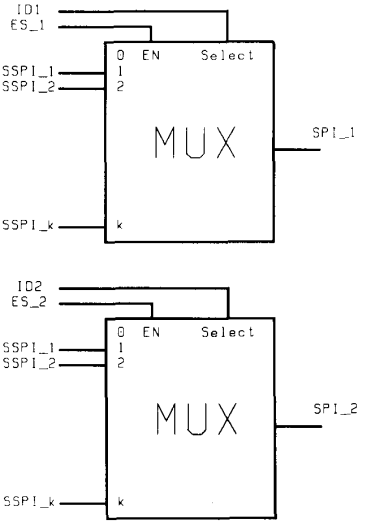


Fig. 4. Switch$_{in}$ basic design.

b) *Switch$_{in}$:* This logic control unit establishes proper links from the outputs of the pipelined nonfaulty processors preceding the faulty ones, to the inputs of the spare processors (SPI_1, SPI_2). The basic logic design of this unit is shown in Fig. 4.

c) *Switch$_{out}$:* This logic control unit establishes proper links from the outputs of the spare processors (SPO_1, SPO_2) back to inputs of the nonfaulty pipelined processors succeeding the faulty ones. The basic logic design of this unit is shown in Fig. 5. In this design, when one of the demultiplexer (DEMUX's) is disabled, all its outputs are equal to 0. Also, the shown OR gate symbols represent banks of OR gates.

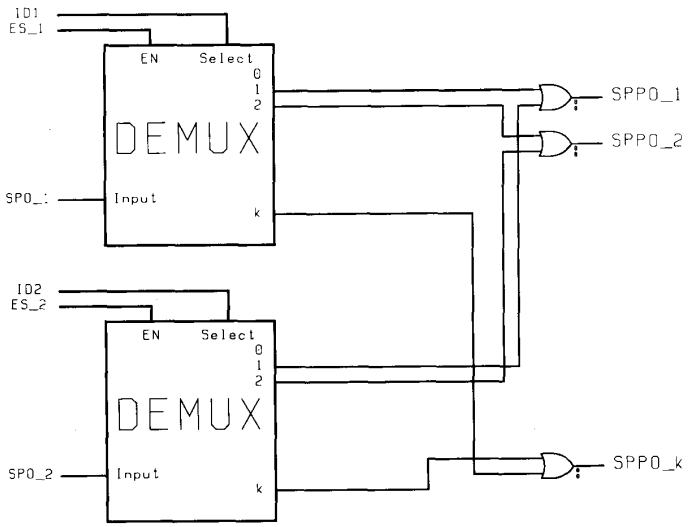


Fig. 5. $\text{Switch}_{\text{out}}$ basic design.

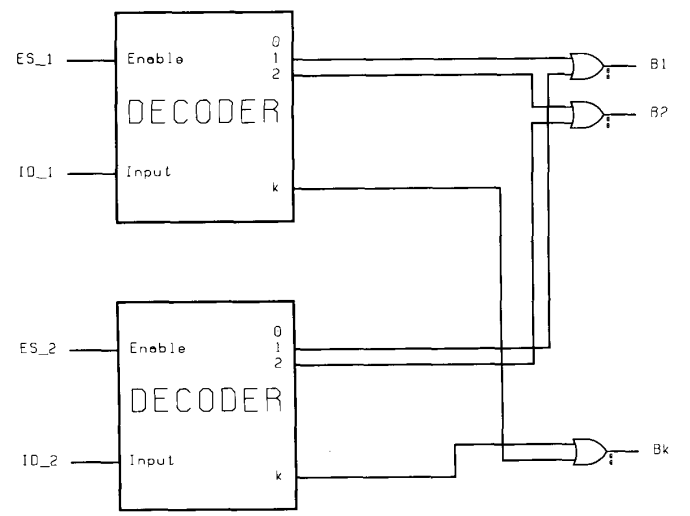


Fig. 6. Selection (bypass) generator basic design.

d) *Selection generator:* This combinational logic unit is used to generate selection signals (B_i, where $i = 0,1,\ldots,k$), which are used in turn by the steering logic at the input of each pipelined processor to reroute the tokens intended for the faulty processor to the spare processor. The steering logic is simply a MUX that is set to appropriately control the flow of information through the pipeline before and after the reconfiguration process. The basic design of this unit is shown in Fig. 6.

e) *Spare processors:* These processors are identical to the processors used in the pipeline. Whenever one or two faults are detected, they can be used to replace the faulty processors.

f) *Pipeline input unit (PIU):* This unit controls the inputs to the pipeline, depending on the state of the system. In normal operation, it supplies image information to the pipelined processors, while during recovery from one or two faults, it prohibits the input from the image memory (IM) to the system.

g) *Software switcher:* This unit is responsible for downloading spare processors with the states and programs of faulty processors in order to take over their role in the pipeline after recovery. It mainly consists of a general-purpose processor and an attached memory. The processor is interrupted by the recovery signal (R) issued by SVC once recovery from a fault is required. The interrupt service routine (ISR) performs the necessary downloading. The attached memory stores a duplicate of the programs of all pipeline processors. Also, it contains a look-up table for storing the starting and ending addresses of each

program. The algorithm of ISR is detailed as follows.

```
1) read ES1, ES2, ID1, ID2
2) FOR i = 1 TO no_of_spares DO
   IF (used_spares = i − 1) AND (ES_i = 1) THEN
   BEGIN
     read lookup [ID_i, 1], lookup [ID_i, 2]
     read memory segment starting at lookup
       [ID_i, 1] and ending at lookup [ID_i, 2]
     write memory segment into SP_i
     write state of faulty processor (ID_i) into SP_i
     used_spares = used_spares + 1
   END
```

In the preceding algorithm, the variable "*used_spares*" is initialized with 0 upon system power up. Moreover, the algorithm could be easily modified for recovery from m number of faults by initializing "*no_of_spares* with the m (in our system, $m = 2$).

The system employs software redundancy, by storing a duplicate of all pipelined processors programs in the attached memory, to achieve the required downloading during recovery. Also, the recovery period is the time required to read the programs of faulty processors and then write them, along with the states of the faulty to the spare processors.

h) *Image memory (IM):* This unit consists of two memory buffers, each of which is capable of storing the whole image. This buffering scheme is employed to maintain concurrency of operations between the camera, which is scanning the scene and storing its image into one buffer, and the PIU, which is reading the previously scanned frame from the other buffer. The camera and PIU operations are switched between the buffers every frame.

B. Rippling Replacement Architecture

The rippling replacement architecture proposed for the NPU is shown in Fig. 3, where the following basic units are identified.

a) *State verification controller (SVC):* This unit has the same function and design of the corresponding unit in Fig. 2.

b) *Bypass generator:* This combinational logic unit is responsible for the generation of bypass control signals (B_i, where $i = 0,1,\ldots,k$) used by the steering logic at the output of each pipelined processor to bypass that processor when it is detected as faulty. The basic design of this unit is mainly the same as that given in Fig. 5.

c) *Pipelined input unit (PIU):* This unit has the same dual function of the corresponding unit in Fig. 2, with the exception that during recovery from one or two faults, it not only prohibits the input from IM but also passes downloading tokens supplied by the software switcher to the pipeline to provide proper recovery.

d) *Software switcher:* The function of this unit is to download each of the processors succeeding the faulty processor in the pipeline with the state and program of its preceding processor. This is accomplished by sending command tokens to the latter in order to pass its state to its successor. This mechanism is employed to maintain a high degree of parallelism during recovery, and it is applied to all processors succeeding the faulty, except the one immediately succeeding it, which must be downloaded with the entire state and program of the faulty processor.

This unit is similar in configuration to the corresponding unit in Fig. 2, although its ISR is different due to the difference between the replacement strategies employed. Also, its attached memory contains, in addition to the programs and look-up table, a download table that contains the necessary command tokens to initiate the downloading of each processor's state and program to its successor. The algorithm of the ISR is detailed as follows.

```
read ES1, ES2, ID1, ID2
FOR i = 1 TO no_of_spares DO
 IF (used_spares = i − 1) AND (ES_i = 1) THEN
 BEGIN
   FOR j = k DOWN TO ID_i + 1 DO
   BEGIN
     read download [ID_j]
     send download [ID_j] to PIU
   END
   read lookup [ID_i, 1], lookup [ID_i, 2]
   read memory segment starting at lookup [ID_i, 1]
     and ending at lookup [ID_i, 2]
   write memory segment into processor number
     ID_i + 1
   write state of faulty processor (ID_i) into processor
     number ID_i + 1
   used_spares = used_spares + 1
 END
```

In the preceding algorithm, the software redundancy employed is more than that of the direct replacement strategy architecture due to the downloading routines added to the program of each pipelined processor, enabling it to download its state to its successor.

The recovery period is the time required not only to read the programs of the faulty processors and write them to their successors, but also the time required to send the download table entries. Hence the recovery period for rippling replacement architecture is longer than that of the direct replacement architecture. This conclusion is in compliance with the fact that the rippling replacement architecture is using less hardware in the switching mechanism, and therefore it must lose some of the speed advantage of hardware switching.

e) *Image memory (IM):* This unit has the same configuration as the corresponding unit in Fig. 2.

C. Switching Mechanism

In both architectures, the switching mechanism is defined as the combined set of coordinated actions performed by the different units of the system to achieve the required switching at the beginning and end of the recovery period. This mechanism is illustrated using the following algorithmic approach.

1) WHILE (NO fault is detected) DO
   BEGIN
   a) SVC scans and tests processors' states
   b) PIU reads image tokens from IM and feeds them to the pipeline
   END
2) SVC
   a) generates *ID* code(s) of failing processor(s)
   b) activates ES_1, or ES_2, or both
   c) activates recovery signal (R)
3) PAR BEGIN:
   a) PIU prohibits input to the pipeline
   b) proper B_i's are generated by the selection (or bypass) generator
   c) software switcher reads code(s) of the faulty processor(s)
   PAR END
4) CASE replacement strategy employed is
   direct replacement:
   BEGIN
   software switcher
   a) reads program of faulty processor(s)
   b) downloads the state and program of faulty to the spare(s)
   END
   rippling replacement:
   BEGIN
   software switcher
   a) issues command tokens to initiate shift of functions operation
   b) reads program of faulty processor(s)
   c) downloads the state and program of the faulty to its successor(s)
   END
   END CASE
   SVC deactivates R, i.e., GOTO 1.

In this mechanism, the combined hardware/software approach of switching is revealed by the interaction of the hardware components, i.e., SVC, PIU, selection (or bypass) generator, $\text{switch}_{\text{out}}$, $\text{switch}_{\text{in}}$, and the steering logic, with the software components, i.e., the software switcher and its attached memory. This combination makes use of the advantages of both hardware switching, i.e., speed and software switching, i.e., noncomplex switching circuitry.

IV. Evaluation of Fault-Tolerant Architectures

The direct and rippling replacement architectures that were discussed earlier achieve the same degree of fault tolerance as opposed to the nonredundant system, which has no fault tolerance at all, i.e., it fails whenever a fault condition exists. This fault tolerance advantage is achieved at the expense of additional hardware and software needed in both redundant architectures. The hardware redundancy ratio in the rippling replacement architecture is less than that of the direct replacement architecture. Conversely, the software redundancy ratio is higher in the rippling replacement architecture.

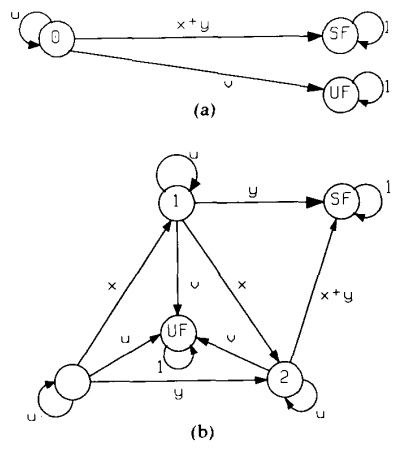


Fig. 7. Markov modes.

As far as the delay of the redundant architectures is concerned, the direct replacement architecture introduces a delay, as compared to the nonredundant system, given by

$$[4+3r]d$$

where

- r 1, if at least one spare is used,
- r 0, if no spares are used,
- d one logic level delay.

The first portion of this delay, *4d*, accounts for the steering logic delays encountered at the input of the faulty processor and at the input of the spare processor, while the second portion, *3rd*, accounts for $\text{Switch}_{\text{out}}$ delay, which is encountered only if one of the spares is used. As for $\text{Switch}_{\text{in}}$ delay, it is not counted since its *2d* delay is accounted for by bypassing the steering logic located before faulty processors.

In the rippling replacement architecture, a constant delay of value *2d* is added independent of the number of spares used, and this delay is due to the steering logic encountered at the output of each processor, including the spare processors.

To evaluate the reliability, availability, and safety of the redundant architectures, as compared to the nonredundant system, two Markov models are developed in Fig. 7.

The first model, shown in Fig. 7(a) is for the nonredundant system, and the second model, in Fig. 7(b) applies for both redundant architectures, since they both employ a standby sparing approach that differs only in the re-

placement strategy. In both models, it is assumed that no more than two faults could happen simultaneously and that the faults are independent. In addition, the following definitions hold for both models:

- 0 state where no spares are used,
- 1 state where 1 spare is used,
- 2 state where 2 spares are used,
- UF state where one or two faults happened without being detected,
- SF state where more than two faults happened, simultaneously, and have been detected,
- λ processor failure rate,
- C Fault coverage factor,

In both models given in Fig. 7, the processors are assumed to obey the exponential failure law with a constant failure rate λ. Therefore the proability of a processor to fail at time $t+\Delta t$, given that it was not faulty at time t, is given by

$$1-e^{-\lambda \Delta t}$$

which could be approximated by $\lambda \Delta t$ for very small values of Δt. Hence the following hold at any state in Fig. 6:

$$x = kC\lambda \Delta t \tag{1}$$

$$y = 0.5k(k-1)(C\lambda \Delta t)^2 \tag{2}$$

$$u = 1 - k\lambda \Delta t - 0.5k(k-1)(\lambda \Delta t)^2 \tag{3}$$

$$v = (1-C)k\lambda \Delta t + 0.5k(k-1)(1-C^2)(\lambda \Delta t)^2 \tag{4}$$

where

- x probability that any of the k processors will fail and be detected,
- y probability that any two of the k processors fail simultaneously and be detected,
- u probability that all processors are working fault-free,
- v probability that any one or two processors fail without being detected.

A. Analysis of the Nonredundant System

It is assumed that the nonredundant system has only a fault detection capability to warn against faults. In the case of faults, operations are suspended and repair personnel are called for service.

From Fig. 7(a), the following equations are obtained for the probabilities of the system being at any state at time $t+\Delta t$

$$p_0(t+\Delta t) = \left[1 - k\lambda \Delta t - 0.5k(k-1)(\lambda \Delta t)^2\right] p_0(t) \tag{5}$$

$$p_{SF}(t+\Delta t) = \left[kC\lambda \Delta t + 0.5k(k-1)(C\lambda \Delta t)^2\right] p_0(t) + p_{SF}(t) \tag{6}$$

$$p_{UF}(t+\Delta t) = \left[(1-C)k\lambda \Delta t + 0.5k(k-1)\cdot(1-C^2)(\lambda \Delta t)^2\right] p_0(t) + p_{UF}(t). \tag{7}$$

The previous equations could be rearranged to yield the following expressions

$$[p_0(t+\Delta t) - p_0(t)]/\Delta t = -k\lambda[1+0.5(k-1)\lambda \Delta t]\, p_0(t)$$

$$[p_{SF}(t+\Delta t) - p_{SF}(t)]/\Delta t = kC\lambda[1+0.5(k-1)C\lambda \Delta t]\, p_0(t)$$

$$[p_{UF}(t+\Delta t) - p_{UF}(t)]/\Delta t = (1-C)k\lambda[1+0.5(k-1)(1+C)\lambda \Delta t]\, p_0(t).$$

By taking the limit as Δt approaches the zero, to arrive at the continuous Markov model equations the following expressions are obtained

$$dp_0(t)/dt = -k\lambda p_0(t) \tag{8}$$

$$dp_{SF}(t)/dt = kC\lambda p_0(t) \tag{9}$$

$$dp_{UF}(t)/dt = (1-C)k\lambda p_0(t) \tag{10}$$

which could be solved assuming the following initial conditions

$$p_0(0) = 1, \qquad p_{UF}(0) = p_{SF}(0) = 0 \tag{11}$$

producing the following final expressions for the probabilities at any time t

$$p_0(t) = e^{-k\lambda t} \tag{12}$$

$$p_{SF}(t) = C(1-e^{-k\lambda t}) \tag{13}$$

$$p_{UF}(t) = (1-C)(1-e^{-k\lambda t}). \tag{14}$$

B. Analysis of the Redundant System

From Fig. 7(b), the following equations are obtained for the probabilities of the system being at any state at time $t+\Delta t$:

$$p_0(t+\Delta t) = \left[1 - k\lambda \Delta t - 0.5k(k-1)(\lambda \Delta t)^2\right] p_0(t) \tag{15}$$

$$p_1(t+\Delta t) = kC\lambda \Delta t\, p_0(t) + \left[1 - k\lambda \Delta t - 0.5k(k-1)(\lambda \Delta t)^2\right] p_1(t) \tag{16}$$

$$p_2(t+\Delta t) = 0.5k(k-1)(C\lambda \Delta t)^2 p_0(t) + kC\lambda \Delta t p_1(t) + \left[1 - k\lambda \Delta t - 0.5k(k-1)(\lambda \Delta t)^2\right] p_2(t) \tag{17}$$

$$p_{SF}(t+\Delta t) = 0.5k(k-1)(C\lambda \Delta t)^2 p_1(t) + \left[kC\lambda \Delta t + 0.5k(k-1)(C\lambda \Delta t)^2\right] p_2(t) + p_{SF}(t) \tag{18}$$

$$p_{UF}(t+\Delta t) = \left[(1-C)k\lambda \Delta t + 0.5k(k-1)(1-C^2)\cdot(\lambda \Delta t)^2\right][p_0(t) + p_1(t) + p_2(t)] + p_{UF}(t). \tag{19}$$

These equations could be rearranged to yield the follow-

ing expressions:

$$[p_0(t+\Delta t)-p_0(t)]/\Delta t = -k\lambda[1+0.5(k-1)\lambda\Delta t]p_0(t)$$

$$[p_1(t+\Delta t)-p_1(t)]/\Delta t = kC\lambda p_0(t)-k\lambda[1+0.5(k-1)\lambda\Delta t]p_1(t)$$

$$[p_2(t+\Delta t)-p_2(t)]/\Delta t = 0.5k(k-1)(C\lambda)^2\Delta t p_0(t)+kC\lambda p_1(t) - k\lambda[1+0.5(k-1)\lambda\Delta t]p_2(t)$$

$$[p_{SF}(t+\Delta t)-p_{SF}(t)]/\Delta t = 0.5k(k-1)(C\lambda)^2\Delta t p_1(t) + kC\lambda[1+0.5(k-1)C\lambda\Delta t]p_2(t)$$

$$[p_{UF}(t+\Delta t)-p_{UF}(t)]/\Delta t = (1-C)k\lambda[1+0.5(k-1)(1+C)\lambda\Delta t]\cdot[p_0(t)+p_1(t)+p_2(t)].$$

And by taking their limit as Δt approaches the zero, to arrive at the continuous Markov model equations the following expressions are obtained:

$$dp_0(t)/dt = -k\lambda p_0(t) \quad (20)$$

$$dp_1(t)/dt = kC\lambda p_0(t)-k\lambda p_1(t) \quad (21)$$

$$dp_2(t)/dt = kC\lambda p_1(t)-k\lambda p_2(t) \quad (22)$$

$$dp_{SF}(t)/dt = kC\lambda p_2(t) \quad (23)$$

$$dp_{UF}(t)/dt = (1-C)k\lambda[p_0(t)+p_1(t)+p_2(t)] \quad (24)$$

which could be solved using the following initial conditions:

$$p_0(0)=1, \qquad p_1(t)=p_2(t)=p_{SF}(0)=p_{UF}(0)=0 \quad (25)$$

producing the following final expressions for the probabilities at any time t:

$$p_0(t)=e^{-k\lambda t} \quad (26)$$

$$p_1(t)=kC\lambda t e^{-k\lambda t} \quad (27)$$

$$p_2(t)=(kC\lambda t)^2 e^{-k\lambda t}/2 \quad (28)$$

$$p_{SF}(t)=C^3\left[1-e^{-k\lambda t}-k\lambda t e^{-k\lambda t}-(k\lambda t)^2 e^{-k\lambda t}/2\right] \quad (29)$$

$$p_{UF}(t)=(1-C)\left[(1+C+C^2)(1-e^{-k\lambda t}) - kC\lambda(1+C)te^{-k\lambda t}-(kC\lambda t)^2 e^{-k\lambda t}/2\right]. \quad (30)$$

C. Reliability

The reliability of a system is defined as the probability that the system is still performing correctly after a period of time t, and is given by a) for the nonredundant system:

$$R(t)=p_0(t)=e^{-k\lambda t} \quad (31)$$

b) for both redundant systems:

$$R(t)=p_0(t)+p_1(t)+p_2(t) = \left[1+kC\lambda t+(kC\lambda t)^2/2\right]e^{-k\lambda t}. \quad (32)$$

Hence an improvement in reliability is achieved by introducing redundancy as compared to the nonredundant system.

D. Availability

The availability $A(t)$ of the system is defined as the probability that the system is performing correctly at a particular time t. The steady-state availability A_{ss} is the value of $A(t)$ as t approaches ∞, and is given by a) for the nonredundant system:

$$A_{ss}=\text{MTTF}/[\text{MTTF}+\text{MTTR}_{nr}] \quad (33)$$

b) for the direct replacement system:

$$A_{ss}=\text{MTTF}/[\text{MTTF}+\text{MTTR}_{dr}] \quad (34)$$

c) for the rippling replacement system:

$$A_{ss}=\text{MTTF}/[\text{MTTF}+\text{MTTR}_{rr}] \quad (35)$$

where

1) MTTF mean time for any processor to fail,
2) MTTR_{nr} mean time to repair the nonredundant system, which is equal to the time it takes to call for service and get the system repaired,
3) MTTR_{dr} mean time to repair the direct replacement system, which is given by the time it takes to download the spare processors,
4) MTTR_{rr} mean time to repair the rippling replacement system, which is given by the time it takes to download all processors succeeding the faulty processor in addition to the spare.

Since $\text{MTTR}_{nr} \gg \text{MTTR}_{dr} > \text{MTTR}_{rr}$, then the introduced redundancy increased the steady-state availability of the system, with a slight improvement in the direct replacement system as compared to the rippling replacement system.

E. Safety

The safety of our system is defined as the probability that either all k processors are operating in a correct manner, or, if one or two faults happen, they are detected, located, and appropriate action is taken either by replacing faulty processors by spare ones or by suspending operations if the two spares have been used previously in place of other faulty processors. The safety $S(t)$ is given by a) for the nonredundant system:

$$S(t)=p_0(t)+p_{SF}(t)=C+(1-C)e^{-k\lambda t} \quad (36)$$

(hence the steady-state safety $[S(\infty)]$ is given by C); b) for

the redundant systems:

$$S(t) = p_0(t) + p_1(t) + p_2(t) + p_{SF}(t)$$
$$= C^3 + \left[(1-C^3) + kC\lambda(1-C^2)t + (kC\lambda)^2(1-C)t^2/2\right]e^{-k\lambda t} \quad (37)$$

(hence the steady-state safety $S(\infty)$ is given by C^3). This expression for the steady-state safety applies for both redundant systems since they possess the same Markov model.

The previous expressions indicate an improved steady-state safety of the redundant systems as compared to the nonredundant system.

V. Feasibility Study

In both fault-tolerant designs discussed in this paper, a pipelined architecture was used to meet the time constraint of real-time image processing.

In this section, the feasibility of implementing those designs using off-the-shelf image processors is studied. The timing analysis of the NPU algorithms is performed to find an estimate for the number of processors required to satisfy the said time constraint. Any digital processor could be used for our system as long as it satisfies the requirements of high-speed image processing and pipelined architecture.

A. NPU Multiprocessor System Configuration

The NPU is configured by cascading several processors in a pipeline to enhance the processing speed. Processing routines partitioning is used to make use of the modular structure of the NPU.

Three main considerations must be taken into account while implementing the configuration described in the preceding.

1) *Synchronization:* This consideration is required for processing routines-partitioning strategies only, in which faster processors must be prevented from overwhelming slower processors by restricting or prohibiting input from outside the slower processor whenever more than a certain number of levels accumulates in its internal queue. An alternative for achieving this consideration is to operate the whole pipeline at one transfer rate.
2) *Processing burden distribution:* If the processing burden varies from one processor to the other, the overall system performance will be greatly influenced by the speed of the most heavily burdened processor. Hence care should be taken to distribute the processing burden among the processors as evenly as possible.
3) *Interfacing to the host:* A pipelined processor is usually designed as a peripheral processor and is normally accessed from the host by performing read/write operations to it as an I/O device. However, external circuitry is required to decode the I/O operations intended for the pipelined processors and resolve the differences in data tokens lengths if they exist.

B. Timing Analysis

In performing the timing analysis of our pipelined system, the following considerations were taken into account.

a) The system must be capable of processing the image within the real-time limit of scanning 30 images per second, which is 33.3 ms.
b) The transfer rate of pixels through the pipeline must be matching for all processors in order to avoid having a bottle neck at the slowest processor that could affect both synchronization and processing burden distribution considerations.
c) The processing burden of the pipeline does not include image processing only but also error detection. Error detection is accomplished locally within each processor by running its own self-diagnostics whenever it is not performing image processing functions.
d) The AT&T WE DSP16A was used as an example of a high-speed DSP in our timing analysis. The DSP16A features a high-speed instruction cycle of 25 ns, in addition to an internal 4096×16 ROM and 2048×16 RAM. Also, each could be expanded to $64k \times 16$ locations externally.

1) Segmentation: The moment-preserving segmentation algorithm proposed by Tsai, in [31], is used. The algorithm could be divided into the following steps.

Moment calculation: The ith moment of an image of size MN is given by

$$m_i = (1/\mathrm{MN}) \sum_x \sum_y f^i(x,y), \qquad i = 1,2,3,4,5 \quad (38)$$

where $f(x,y)$ denotes the image function value at pixel (x,y).

The moment calculation is divided among five processors, one for each moment. The time devoted for one pixel by each processor is the time required to perform the following operations.

1) Input the pixel value and the previous product from the preceding processor.
2) Multiply the pixel value by the previous product.
3) Accumulate the multiplication product.
4) Output the multiplication product to the succeeding processor.

In the DSP16A, concurrent execution of arithmetic and transfer operations is feasible. Hence the preceding four operations could be performed in two instruction cycles, with an overhead of 50 ns. Two noteworthy points are that no overhead is incurred for instructions "fetch" if they were first transferred into the processor internal cache; and that the I/O operations are done as memory

references to ROM/RAM locations outside the address space of the internal ROM and RAM.

2) Threshold Levels Calculation: This step is performed by first calculating a set of auxiliary values (c_0, c_1, c_2), which are related to the moments by the following set of equations:

$$c_0 m_0 + c_1 m_1 + c_2 m_2 = -m_3 \tag{39}$$

$$c_0 m_1 + c_1 m_2 + c_2 m_3 = -m_4 \tag{40}$$

$$c_0 m_2 + c_1 m_3 + c_2 m_4 = -m_5 \tag{41}$$

The time required for this step is given by $20t_a + 35t_m + 3t_d$, where t_a, t_m, and t_d are the times required to perform addition, multiplication, and division, respectively.

Then the following polynomial equation is solved to obtain the representative gray levels (z_0, z_1, z_2):

$$z^3 + c_2 z^2 + c_1 z + c_0 = 0. \tag{42}$$

The time required for this step is given by $12t_a + 12t_m + 11t_d + 2t_p$, where t_p is the time required to perform exponentiation.

And eventually the threshold levels (p_0, p_1, p_2) are calculated by solving the following moment-preserving equations:

$$p_0 z_0^0 + p_1 z_1^0 + p_2 z_2^0 = m_0 \tag{43}$$

$$p_0 z_0^1 + p_1 z_1^1 + p_2 z_2^1 = m_1 \tag{44}$$

$$p_0 z_0^2 + p_1 z_1^2 + p_2 z_2^2 = m_2. \tag{45}$$

The time required for this step is given by $17t_a + 32t_m + 2t_d$. Therefore the total time required to calculate the thresholds is given by

$$Th = 49t_a + 79t_m + 16t_d + 2t_p. \tag{46}$$

If each addition, multiplication, and division is performed in one instruction cycle and the exponentiation is performed in five instruction cycles, this is reduced to 0.00385 ms.

Hence the total time required for moment and threshold-level calculations is given by

$$T_1 = [q + (MN - 1)]\, p_t + Th \tag{47}$$

where

- q number of stages in the moment calculation pipeline, i.e., $q = 5$,
- p_t pixel transfer period, i.e., $p_t = 50$ ns,
- M number of scan lines in the image,
- N number of pixels in each scan line.

For an image of size 512×512, with pixels of 16 gray levels (which is adequate for an application that differentiates between three different classes of objects, paths, barcodes, and background), T_1 is reduced to 13.13 ms.

The time for both moment and threshold calculations were added because they are performed sequentially, since threshold levels can not be calculated unless moments are available. Hence threshold-levels calculation could be performed by the processor used for the fifth moment, after it finishes moment calculation. Also, the rest of the system must wait for the thresholds to be calculated in order to start the image classification, smoothing, thinning, and finally node/edge extraction. This waiting time is utilized in performing internal self-diagnostics within each processor for the purpose of error detection.

3) Pixels Classification: This step is performed by one processor that rescans the image pixels and classifies them according to the previously calculated thresholds. This is accomplished by the input of the image pixel, its comparison with the thresholds, and the output of its new value. The output value for the NPU is 1 for paths and 0 for barcodes and background. However, for the BPU it is 1 for barcodes and 0 for paths and background. The two output values could be generated simultaneously on two different pins of the data bus. Hence the pixel classification of each pixel is carried out in two instruction cycles, with an execution time of 50 ns.

4) Smoothing: The smoothing algorithm used in [29] and [30] is based on the algorithm proposed in [32]. Smoothing is done through three different steps performed by three successive processors

1) In the first step, the algorithm fills in small (one pixel) holes in dark areas, and small notches in straightedge segments. The algorithm evaluates the following Boolean expression, to accomplish the said functions, for each pixel p and its eight connected neighbors, a through h:

$$B_1 = p + b \cdot g \cdot (d + e) + d \cdot e \cdot (b + g). \tag{48}$$

Then, if $B_1 = 1$, the pixel p is assigned 1; otherwise it is assigned 0.

Since the evaluation of B_1 requires the availability of its eight-connected neighbors, the shift buffer proposed in [33] is used. To speed up the time required to evaluate B_1, a look-up table stored in the processor internal RAM is implemented. The look-up table has 32 entries for all combinations of p, b, d, e, and g. The address of each entry is pointed at by their values, and its contents are the values of B_1. Hence two instruction cycles are required to perform the first step for each pixel; one for reading the pixel value from the previous processor, and the other is for reading the value of B_1 stored in the look-up table.

2) In the second step, the algorithm eliminates isolated 1's, and small bumps along straight-edge segments. This is accomplished by evaluating the following expression for all pixels:

$$B_2 = p \cdot [(a + b + d) \cdot (e + g + h) + (b + c + e) \cdot (d + f + g)]. \tag{49}$$

Then, if $B_2 = 1$, the pixel p is assigned 1; otherwise it is assigned 0.

3) In the third and last step, the algorithm replaces missing corner points by evaluating the following set of Boolean expressions to check for top right corner, bottom right corner, top left corner, and bottom left corner points:

$$B_3 = p + \bar{p} \cdot \left[(d \cdot f \cdot g) \cdot \overline{(a + b + c + e + h)} \right] \quad (50)$$

$$B_4 = p + \bar{p} \cdot \left[(a \cdot b \cdot d) \cdot \overline{(c + e + f + g + h)} \right] \quad (51)$$

$$B_5 = p + \bar{p} \cdot \left[(e \cdot g \cdot h) \cdot \overline{(a + b + c + d + f)} \right] \quad (52)$$

$$B_6 = p + \bar{p} \cdot \left[(b \cdot c \cdot e) \cdot \overline{(a + d + f + g + h)} \right]. \quad (53)$$

Then, if B_3, B_4, B_5, or $B_6 = 1$, pixel p is assigned 1; otherwise it is assigned 0.

The evaluation of the new pixel value at the end of the second and the third steps of smoothing could be done similar to the first step, except that the look-up tables used must have 512 entries to accommodate all possibilities of p and its eight connected neighbors. Also, in the third step, one look-up table is used to store the logical OR-ing of B_3 through B_6.

5) Thinning: The algorithm used for thinning in [29] and [30] is the same one proposed in [34]. In this algorithm a number of passes r, which depends on the input image, is done to obtain the thinned image. In each pass the following expressions are calculated at most for all pixels:

$$S_0 = d \cdot (a + b + f + g) \cdot (g + \bar{h}) \cdot (b + \bar{c}) \quad (54)$$

$$S_2 = g \cdot (d + e + f + h) \cdot (\bar{c} + e) \cdot (\bar{a} + d) \quad (55)$$

$$S_4 = e \cdot (b + c + g + h) \cdot (\bar{a} + b) \cdot (\bar{f} + g) \quad (56)$$

$$S_6 = b \cdot (a + c + d + e) \cdot (d + \bar{f}) \cdot (e + \bar{h}). \quad (57)$$

Then, if a pixel p is originally equal to 1, and S_0, S_2, S_4, or $S_6 = 1$, pixel p is assigned 0. This operation is performed iteratively for the whole image until no pixel values are changed. The value of p after performing the thinning operation could be evaluated using twelve processors (for edges that require twelve passes to be thinned). The value of p at the end of each pass is evaluated using a look-up table similar to that used in smoothing.

Hence the total time required to perform image classification, smoothing, and thinning using w ($w = 16$) processors is given by

$$T_2 = [3wM + (M - 3)N]\, p_t = 14.26 \text{ ms}. \quad (58)$$

The first portion $3wMp_t$ is spent in propagating the first three scan lines through the pipeline, starting at the pixel classification processor, until they reach the last thinning processor, while the second portion $(M-3)Np_t$ is used for the processing of the rest of the image. During the time T_2, the moment calculation processors are running their internal self-diagnostics for error detection.

6) Node/Edge Extraction: The NPU performs an off-line learning of the map of the path network before it starts navigation. Then, during navigation, the AGV scans its surroundings through a camera, detects its current position by monitoring a reference path, painted on the ground or the ceiling, and the bar codes attached to the network nodes, and then it verifies this information against the information previously stored during the learning phase. The time required for node/edge extraction is mainly dependent on the number of nodes that appear in the image and the length of their interconnecting edges, which in turn depends on the resolution and range of the used camera. In the worst case, the AGV can see the whole path network during navigation, i.e., it processes the whole path network image that was processed off-line. The node/edge extraction algorithm used is detailed in the Appendix. The algorithm consists of two steps; initialization and edge-tracing. The time required for initialization is given by T_3 and that for edge tracing is given by T_4:

$$T_3 = (n + 6)t_A + 10\, x\, t_C \quad (59)$$

$$T_4 = e[(b + 4)t_A + (19b + 16)t_C] \quad (60)$$

where

- e maximum number of edges in the input image,
- x number of pixels tested before the first crossing node is found,
- b maximum length of an edge,
- n maximum number of nodes in the input image (at most $n = e + 1$),
- t_A time required to perform an assignment,
- t_C time required to perform a comparison.

Therefore the total time required for node/edge extraction is equal to 18.4 ms for typical values of $e = 20$, $b = 200$, $x = 25$ percent of all pixels, $t_A = t_C = 25$ ns. Hence only one processor is capable of performing the required node/edge extraction. However, due to the random-access manner in which this processor needs to access image pixels, it requires an attached 32 kB of cache memory to store the image after thinning. Also, it can not start its operations until the last thinning pass is already carried out.

VI. Discussion

It was shown that a total of 22 processors is capable of implementing the NPU. However, to illustrate that they are capable of performing the required image processing, in addition to error detection, and still meet the real-time constraint, scheduling of T_1 through T_4 during each frame is carried out. In Fig. 8, the terms A_i, B_i, and C_i are defined as follows.

- A_i Moments and thresholds calculation of frame i.
- B_i Classification, smoothing, and thinning of frame i.
- C_i Node/edge extraction of frame i.
- S Self-diagnostics used for error detection.

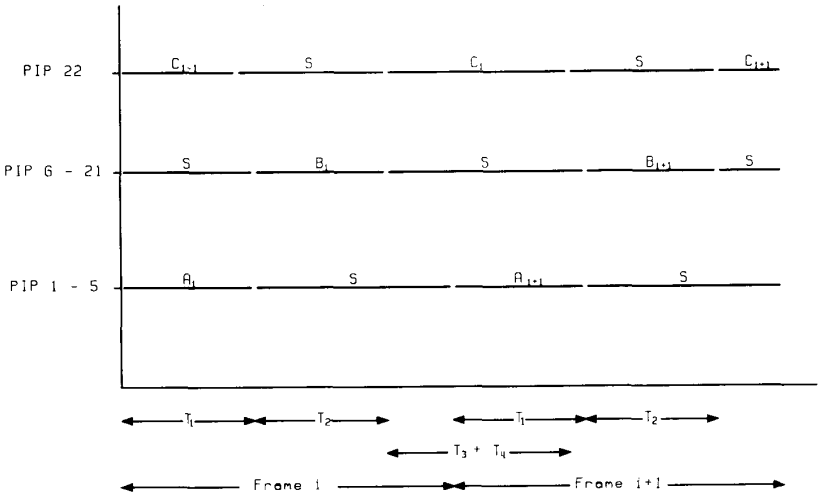


Fig. 8. Scheduling of image processing and error detection.

As shown in Fig. 8, the system makes use of the inherent sequential characteristics of the image processing to provide time slots for performing error detection functions on a frame-by-frame basis. One noteworthy remark is that the additional time available after T_1 and T_2 within each frame can accommodate the delays introduced by adding fault tolerance.

Another alternative for the NPU architecture is to implement the whole pipeline using VLSI technology. The first subsystem of the pipeline is mainly a group of multiply-and-accumulate type of processors used for moment calculation, while the second subsystem used for the rest of the image-processing functions is a group of logical operations processors. Fault tolerance is still applied in this implementation, but on a subsystem basis. This is accomplished by treating each subsystem as a separate pipeline that has its own spare processors.

VII. Conclusion

A fault-tolerant architecture for the AGV system described in [29] and [30] was presented using two different reconfiguration strategies. The reliability, availability, and safety of the proposed architectures were investigated. In addition to this, the relative merits of both architectures were discussed. Also, the feasibility of implementing the proposed system to meet the real-time image processing constraint was studied. It was shown that the node/edge extraction unit is capable of extracting information about nodes and edges in real time. This unit could be used, in a wider context, for implementing any application that requires a transformation of a path network map into an interconnected graph.

Appendix

The algorithm presented in this Appendix is for finding nodes and tracing edges simultaneously. The following assumptions used in this algorithm are that

a) n is the maximum number of nodes an image can have,
b) V is a vector of size n that consists of two components: $V[1 \cdots n]x$ and $V[1 \cdots n]y$,
c) VP is a pointer that initially points to the first location of V minus 1,
d) $E[a, b]$ is an array whose elements are linked lists that store the edge from node a to node b.

```
1) initialize vector V[1 ··· n]x to −1
2) VP = 1
3) go through map image and find a crossing point
   node
4) store node coordinates in first available location of
   vector V
5) VP = VP + 1
6) coordinates of P = coordinates of found node
7) WHILE any of the 8-connected neighbors of P is an
   object point DO
8)       call TRACING_AN_EDGE
9)       IF a new node is found THEN
10)        store new node coordinates in the first avail-
             able location of vector V
11)      END IF
12) END WHILE
13) IF V[VP]x = −1 THEN
14)   RETURN
15)   ELSE
16)   VP = VP + 1
17)   get next node in V (* coordinates of P =
      coordinates of V[VP]*)
18)   GOTO step 7
19) END IF
20) store processed information in memory model
    I(* end of algorithm *)
```

In the previous algorithm calls to the procedure TRACING_AN_EDGE are done to obtain the 8-directional chain code of the edge. This procedure is detailed as follows

1) IF there is an object point Q in P's 8-connected neighbors THEN
2) coordinates of P = coordinates of Q
3) store chain code corresponding to direction from P to Q in a temporary linked list
4) END IF
5) IF P is not a crossing point node OR previous found node THEN
6) delete P from map image
7) ELSE
8) GOTO step 1
9) END IF
10) denote P as a node in the map image
11) store forward chain code linked list in E[initial node, final node]
12) store reversed chain code linked list in E[final node, initial node]
13) RETURN.

REFERENCES

[1] M. H. E. Larcommbe, "Tracking stability of wire guided vehicles," in *Proc. Int. Conf. Auto. Guided Veh. Syst.*, June 1981, pp. 137–144.

[2] Keith C. Drake, E. S. McVey, and R. M. Inigo, "Sensing error for a mobile robot using line navigation," *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. PAMI-7, no. 4, pp. 485–490, July 1985.

[3] E. S. McVey, K. C. Drake, and R. M. Inigo, "Range measurements by a mobile robot using a navigation line," *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. PAMI-8, no. 1, pp. 105–109, Jan. 1986.

[4] T. Tsumura, "Survey of automated guided vehicle in Japanese factory," in *IEEE Int. Conf. on Rob. and Auto.*, 1986, pp. 1329–1334.

[5] A. M. Waxman, J. Lemoigne, L. Davis, B. Srinivasan, T. Kushner, E. Liang, and T. Siddalinggaiah, "A visual navigation system for autonomous land vehicles," *IEEE J. Rob. Auto.*, vol. RA-3, no. 2, pp. 124–141, Apr. 1987.

[6] A. M. Waxman, J. Lemoigne, and B. Srinivasan, "Visual navigation of roadways," in *IEEE Int. Conf. on Rob. Auto.*, Mar. 1985, pp. 862–867.

[7] H. P. Moravec, "Visual mapping by a robot rover," in *Proc. 6th Joint Int. Conf. Artif. Intell.*, Aug. 1979, pp. 589–600.

[8] D. Kuan and U. K. Sharma, "Model-based geometric reasoning for autonomous road following," in *IEEE Int. Conf. on Rob. and Auto.*, Mar. 1987, pp. 416–423.

[9] J. Louise, M. Thomas, K. Gremban, and M. Turk, "The autonomous land vehicle preliminary road following demonstration," in *Proc. SPIE Conf. Computer Vision and Intelligent Robotics*, Cambridge, MA, 1985.

[10] R. Wallace, "Robot road following by adaptive color classification and shape tracking," in *IEEE Int. Conf. on Rob. and Auto.*, 1987, pp. 256–263.

[11] D. Marr, *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*, San Francisco, CA: Freeman, 1982.

[12] R. C. Arkin, E. Riseman, and A. Hanson, "Visual strategies for mobile robot navigation," in *Proc. IEEE Workshop on Computer Vision*, Miami Beach, FL, 1987, pp. 176–181.

[13] R. A. Brooks, "Visual map making for a mobile robot," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Mar. 1985, pp. 824–829.

[14] M. K. Brown, "On ultrasonic detection of surface features," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1785–1790.

[15] M. R. Driels, "Pose estimation using tactile sensor data for assembly operations," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1255–1261.

[16] J. L. Schneiter, "An objective tactile sensing strategy for object recognition and localization," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1262–1267.

[17] S. A. Stansfield, "Primitives, features and exploratory procedures: Building a robot tactile perception system," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1274–1279.

[18] D. Seigel, I. Garabeita, and J. M. Hollerbach, "An integrated tactile and thermal sensor," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1286–1291.

[19] T. Tsumura and M. Hashimoto, "Positioning and guidance of ground vehicle by use of laser and corner cube," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1335–1342.

[20] K. Nishide, M. Hanawa, and T. Kondo, "Automatic position findings of vehicles by means of laser," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1343–1348.

[21] S. Seida, D. Morgenthaler, M. Podlaseck, B. Douglass, J. McSwain, R. Knourek, and M. Thomas, "Vision based road following in the autonomous land vehicle," in *Proc. of the 26th Conf. on Decision and Control*, Los Angeles, CA, Dec. 1987, pp. 1814–1819.

[22] T. Takeda, A. Kato, T. Suzuki, and M. Hossi, "Automated vehicle guidance using spotmark," in *Proc. of IEEE Int. Conf. on Rob. and Auto.*, Apr. 1986, pp. 1349–1354.

[23] D. Alley and R. M. Inigo, "Guided vehicle suboptimal path planning for execution speed in a factory environment," in *Appl. AI VI, SPIE*, Orlando, FL, vol. 937, Apr. 1988, pp. 419–427.

[24] T. Sudkamp, C. Lizza, and C. Wagner, "Hierarchic path generation," in *Appl. AI VI, SPIE*, Orlando, FL, vol. 937, Apr. 1988, pp. 442–449.

[25] P. Levi, "Principles of planning and control concepts for autonomous mobile robots," in *Proc. IEEE Int. Conf. on Rob. and Auto.*, Mar. 1987, pp. 874–881.

[26] V. Lumelsky, "Algorithmic issues of sensor-based robot motion planning," in *Proc. of the 28th Conf. on Decision and Control*, Los Angeles, CA, Dec. 1987, pp. 1796–1801.

[27] P. Siy, "Road map production system for intelligent mobile robots," in *IEEE Int. Conf. on Robotics*, Mar. 1984, pp. 562–570.

[28] D. D. Grossman, "Traffic control of multiple robot vehicles," *IEEE Trans. Robot. Auto.*, vol. 4, no. 5, pp. 491–497, Oct. 1988.

[29] K. Fok and M. Kabuka, "Automatic navigation system for vision-guided vehicles using double heuristic and finite state machine," *IEEE Trans. Robot. Auto.*, accepted for publication.

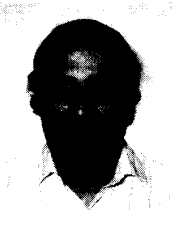
[30] K. Fok and M. Kabuka, "An efficient navigation system for vision guided vehicles," in *Proc. of Florida Artificial Intelligence Conf.*, Orlando, FL, May 1988, pp. 51–58.

[31] W. H. Tsai, "Moment-preserving thresholding: A new approach," *Comp. Vision, Graph. Image Proc.*, vol. 29, pp. 377–393, 1985.

[32] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee, *Robotics: Control, Sensing, Vision, and Intelligence*. New York: McGraw-Hill, 1987.

[33] T. Mimaroglu, "A high-speed two-dimensional hardware convolver for image processing," in *Proc. of IEEE Computer Society Conf. on Pattern Recognition and Image Processing*, 1982, pp. 386–389.

[34] N. J. Naccache and R. Shinghal, "A proposed algorithm for thinning binary patterns," *IEEE Trans. Syst. Man Cyber.*, vol. SMC-14, no. 3, pp. 409–418, 1984.



Mansur R. Kabuka received the B.S. degree in electrical engineering and computer science from the University of Alexandria, the M.S. degree from the University of Miami, Coral Gables, FL, and the Ph.D. degree from the University of Virginia, Charlottesville, VA.

In 1983, he joined the faculty of the Department of Electrical and Computer Engineering of the University of Miami, where he is currently an Associate Professor. During the summer of 1984, he was employed by IBM, Boca

Raton, FL. His research interests include computer vision, special-purpose computer architecture, pattern recognition, and robotics.

Surjadi Harjadi received the B.S. degree (honors) in electrical and computer engineering from the University of Miami, FL, in 1987.

Currently, he is a Teaching Assistant at the University of Miami, FL. He is now working towards the M.S. degree in electrical and computer engineering at the same university. His current research interests include computer vision, artificial intelligence, and robotics.

Mr. Harjadi is a student member of IEEE Computer Society and a member of Eta Kappa Nu and Tau Beta Pi.

Akmal Younis received the B.S. (honors) degree in electrical and computer engineering from Ain Shams University, Cairo, in 1986.

From 1986 to 1987, he was a Teaching Assistant in the Department of Electronics and Computer Engineering, Ain Shams University. In 1988 he joined IBM Cairo Scientific Center. Currently he is working towards the M.S. degree at the University of Miami, FL. His research interests include computer vision and robotics.