MARIGNAN

AN INTERMITTENT FAILURE CORRECTION METHOD

Y. DESWARTE ♦ J. LAVICTOIRE

COMPAGNIE INTERNATIONALE POUR L'INFORMATIQUE

DIVISION MILITAIRE SPATIALE ET AERONAUTIQUE

VÉLIZY 10 Avenue de l'Europe 78140 FRANCE

Abstract

Since intermittent failures are the most frequent to occur in installed hardware (that is, passed the early failure period), it is particularly important to correct errors caused by these intermittent failures.

The Marignan structure attempts to optimize two criteria :
- The efficiency in detecting errors and correcting them, and
- System workload ratio (WLR).

Efficiency is ensured by comparing two executions of the same program before a possible error can be propagated, and by automatic rollback. Double execution can be performed on two different computers (spatial redundancy) or on a single computer (temporal redundancy).

System WLR is optimized by :
- Minimizing checkpoint storage,
- Performing comparisons as infrequently and rapidly as possible,
- Using standard hardware whenever possible,
- Ensuring structure transparency in terms of software.

Introduction

The study of Marignan, Intermittent Failure Correction Method, was conducted under contract DRME 73/ 375. Based on certain analyses, (1.2) intermittent failures would cause 80 % to 90 % of the errors in data processing systems once the installation period is terminated, that is, once the early hardware failures are corrected.

But conventional maintenance (failure detection, localization, and repair) is not really applicable to intermittent failures when they do not frequently occur.
- intermittent failures can be detected only by continuous information surveillance,
- they are localized by trying to get them permanent ; this is accomplished by operating the computer in the power-supply voltage and clock frequency marginal areas.

In the failure localization procedure the computer therefore cannot remain operational.

However, if the system is able to tolerate these failures, processing can continue since the failures are intermittent. As a result, it is more interesting to "mask" the effect of intermittent failures rather than repairing the equipment.

Different methods have been elaborated for this. To compare these methods, we are going to introduce efficiency and performance criteria.

Evaluation Criteria

Efficiency

We define the efficiency of a system that can tolerate intermittent failures as the percentage of errors due to intermittent failures that have been corrected successfully by the system.

Efficiency defined in this way can be evaluated only statistically over a large number of intermittent failures in operational systems.

But if the correction method coverage can be evaluated (that is, the proportion of hardware that this method covers), and if we assume that the covered hardware and the uncovered hardware have the same tendency to intermittent failures, the coverage is a good estimate of the efficiency :

$$\text{Efficiency} \simeq \text{coverage} = \frac{Qc}{Qt}$$

Where :

Qc is the quantity of covered hardware(evaluated, for example, in number gates)
Qt is the total quantity of hardware.

Workload ratio

We define the WLR r of a redundant system R which tolerates intermittent failures with respect to a non-redundant equivalent system 0 by the equation :

$$WLR\ r = \frac{Qo}{QR} \times \frac{To}{TR}$$

Where :

Qo is the quantity of hardware (evaluated, for example, in number of gates) of system 0
QR is the quantity of hardware of system R
To is the execution time of a program given by system 0
TR is the execution time of the same program by system R.

The product QixTi is the " workload" required for system i to execute the program.

Conventional Intermittent Failure Tolerance Methods

Error correcting codes

Coding of a data word is sufficiently redundant to enable correction of any error on a single bit of the word (code SEC - Simple Error Correcting - ex. Hamming code).

The efficiency of this method is limited by :
- the correction of a single bit per word,
- the difficulty in saving the code in arithmetic or logic operations,
- the difficulty in monitoring commands :
  - instruction codes
  - addresses
  - micro-instructions
  - decoding circuits
  - clocks
  - etc...

In general, the efficiency of a central processing unit will not exceed 50 % which makes this method unacceptable for high-reliability systems.

On the other hand, its performance is rather good since the additional " workload" is limited to :
- larger data paths,
- a more sophisticated and slower arithmetic and logic operator,
- test and correction circuits.

WLR can reach 0.7 or 0.8.

Instruction retry

If an error is detected (for example, by detecting code associated with the data) before the initial data of the instruction have been modified, this instruction may be tried to be re-executed.

The efficiency of this method is restricted by :
- detecting code efficiency,
- non-retryable instructions (test-and-set, byte string instruction,...),
- code saving in the arithmetic and logic operator,
- difficulties in monitoring commands ...

Efficiency will therefore still be around 50 %.

For the same reasons as in paragraph "error correcting codes", WLR can reach 0.7 or 0.8.

Triple modular redundancy with a majority vote

The hardware is divided into "functions". Depending on the case, the function can extend from a complex logic circuit to a central processing unit.

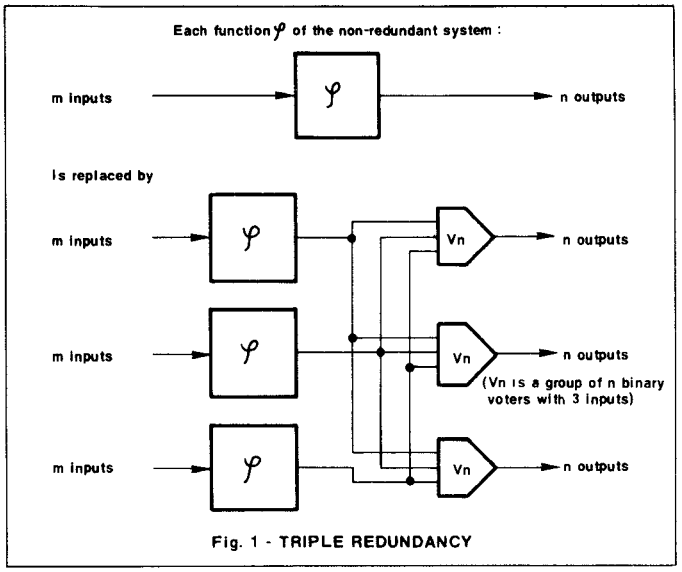

Fig. 1 - TRIPLE REDUNDANCY

This method when generalized for all the system functions permits an efficiency close to 100 % : any intermittent failure affecting an elementary function, a voter, or a link will be masked.

On the other hand, WLR will be less than 0.3. In fact, the quantity of hardware is more than tripled and the execution time is extended.

We will try to show that the Marignan method offers an efficiency also close to 100 % for a WLR close to 0.5.

The Marignan Method Principle

The method consists of executing each processing twice and then comparing the two executions. If their results are different, the program is resumed (execution of a roll-back). This provides a protection against intermittent failures.

Finally, the following have to be defined :
- the comparison level,
- the comparison method,
- the roll-back method,

Comparison level

Comparisons have to be frequent enough to prevent possible error propagation beyond the scope of a roll-back. On the other hand, comparisons should be as infrequent as possible so that system performances are not penalized too much.

The optimal solution is therefore to carry out comparisons just before a possible error would be propagated "outside" the faulty task by modifying data common to the other tasks or by making an I/O request.

Therefore, we will cause a comparison on the results of both executions to be carried out whenever an action to the "outside" occurs, that is :
- at each call or return of a non-re-entrant monitor module,
- at each interrupt acknowledgment,
- at each read direct or write direct,
- at each common data modification,
- whenever the scope of a possible roll-back exceeds a maximum span (see "Roll-back")
- at each trap (mode violation, memory protection, etc...).

This leads us to the notion of "sequence" which we define as the running of a task between two comparisons caused by one of the events above. Note that the sequence is a dynamic entity, transparent to the user, defined only at execution time.

The "sequence" can therefore be introduced as an additional level in the processing hierarchy :

Data Processing System
.
.
.
Task
.
.
.
Sequence
.
.
.
Instruction
.
.
.
Microinstruction

The sequence appears as the largest entity in which an error can be isolated preventing propagation of this error. As a result, we consider it as the optimum comparison level.

Comparison Method

To compare two executions of the same sequence, it is necessary and sufficient to compare the "results" of the sequence, that is, data modified during the sequence. To reduce the comparison cost, only the two arithmetic sums of the results of both executions will be compared ("Check Sum" : CHS). This check-sum is calculated during the sequence for each instruction making a change in memory :

$$CHS = (CHS) + \text{written value} + \text{address}$$

When the sequence is terminated, the value of CHS is incremented by the value of the registers and the program counter (C.O.)

$$CHS = (CHS) + \Sigma_i (Ri) + (C.O.)$$

Roll-Back

When an error is detected in the "end of sequence" comparison, program execution must be resumed. To do this, the machine has to be reset in the "status" where it was before the error occurred, and program execution reinitiated.

Since we know that the error occurred somewhere within the sequence, and no action took place outside the task, execution can and must be resumed at the beginning of the sequence.

To do this, data which might have been modified during execution of the sequence registers, the program status word, and the program counter have to be restored.

A conventional method consists of saving the program context at each checkpoint (here the beginning of the sequence), that is, all memory words liable to be modified before an error is detected. The data to be saved therefore greatly exceeds the data actually modified during the sequence which takes an excessive amount of memory and duplication time. This is why it is generally preferable to let the programmer take care of generating himself his checkpoints explicitly or implicitly (for example, whenever an Algol block is entered,...).

We have chosen to save only memory words actually modified during the sequence. In practice, before a word is written in memory the word that will be destroyed and its address are saved in a save stack associated with the program. If at the end of the sequence both check-sums are different, all the words modified during the sequence can be restored by searching for their value and their address in the stack (beginning with the most recently stacked).

The save area will also contain the registers, the program status word, and the program counter of the beginning of the sequence. The stack size defines the "maximum scope" of the roll-back. Stack saturation causes an "end of sequence".

This general principle can be applied to various architectures : bi-computer, single-processor, multi-computer, or multi-processors.

Marignan Bi-Computer : Spatial Redundancy

The method consists in this case of executing two identical copies of the same program on two identical computers, each one with its own memory, with synchronization of both computers at the end of each sequence.

Hardware structure (see fig. 2)

We will use two Mitra 15s for the example, but the structure can be adapted to any other type of computer.

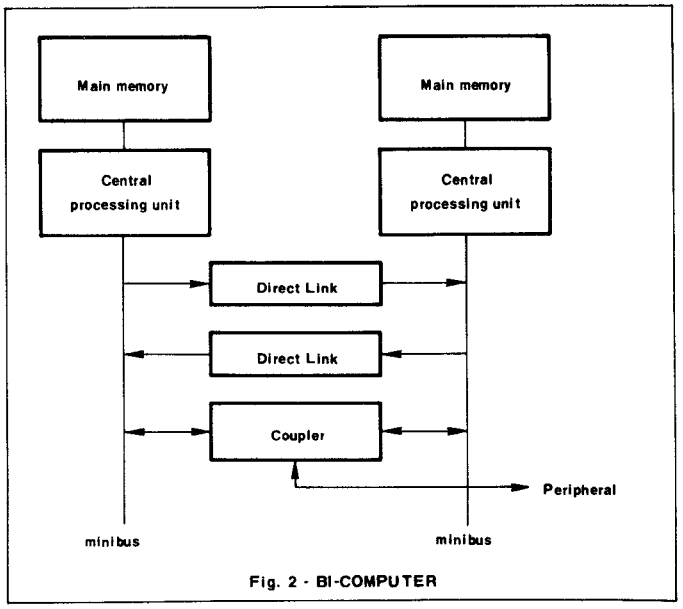

Fig. 2 - BI-COMPUTER

Both computers are interconnected only via a double "direct link" which permits each computer to transfer a word into a register that the other computer can consult.

Since data must be identical in both memories, the couplers are modified as follows :

- in input : data coming from the peripheral is copied on both minibuses with a watchdog synchronization check on the input requests from both computers,
- in output: data coming from both minibuses are compared before being transmitted to the peripheral. A watchdog synchronization check is also performed on the output requests from both computers.

Program structure

A standard version Mitra 15 program has :

- a global data area pointed by the global base used in particular as a data area for re-entrant monitor modules,
- local data areas pointed successively by the local base,
- program areas.

We add to this standard structure simply a save area which is part of the global data, and includes :

- the save stack PS,
- the stack S "top" pointer,
- a save area for the registers, the program status word, and the program counter at the beginning of the sequence.
- the check-sum CHS,
- indicators : IND : Interrupt indication during the " end of sequence " procedure,
  REP : number of roll-backs executed,
  MOT : direct link buffer.

Since this save area is created when the program is loaded, this structure is transparent for the programmer.

In the monitor modules, a special instruction will be added : FDS (End Of Sequence) before each common data modification. The error and trap processing procedures can also be modified to take into account roll-back possibilities.

Microprogram modification

Processing for the redundancy method (check-sum calculation, save, synchronization, comparison, and roll-back) necessitate only microprogram modifications.

Check-sum calculation. The check-sum calculation is added to the microprograms executing the instructions that carry out modifications in memory. To do this, it is necessary to search for the actual check-sum value, add the word that is going to be written in memory along with its address, and store the check-sum.

Save. These memory modification instructions begin by :

- pushing down the word that will be modified along with its address,
- updating the stack top pointer,
- testing save area overflow ; an overflow causes the execution of the " end of sequence (FDS) " microprogram.

End of sequence : FDS. The microprograms processing :

- read or write direct,
- interrupt acknowledgments,
- traps,
- special instruction FDS,
- memory modification instructions when the save stack is saturated call on the microprogram described in fig. 3.

The word transmitted from one computer to the other can have three formats :

- MOT = 0 : current sequence
- MOT = - 1 : correct comparison
- MOT ≠ 0 and ≠ - 1 : check-sum transmitted in "end of sequence".

Therefore, if by chance the check-sum calculated during the sequence is zero or - 1, an arbitrary value will be given to it, for example, + 1. This corresponds to the "formatting" of the check-sum in the flow chart.

Note that in this "end of sequence" procedure, a watchdog (WTCHDG) monitors the synchronization of both computers and also checks that both computers are in agreement for executing a roll-back or not. If they do not agree, the error is declared irrecoverable.

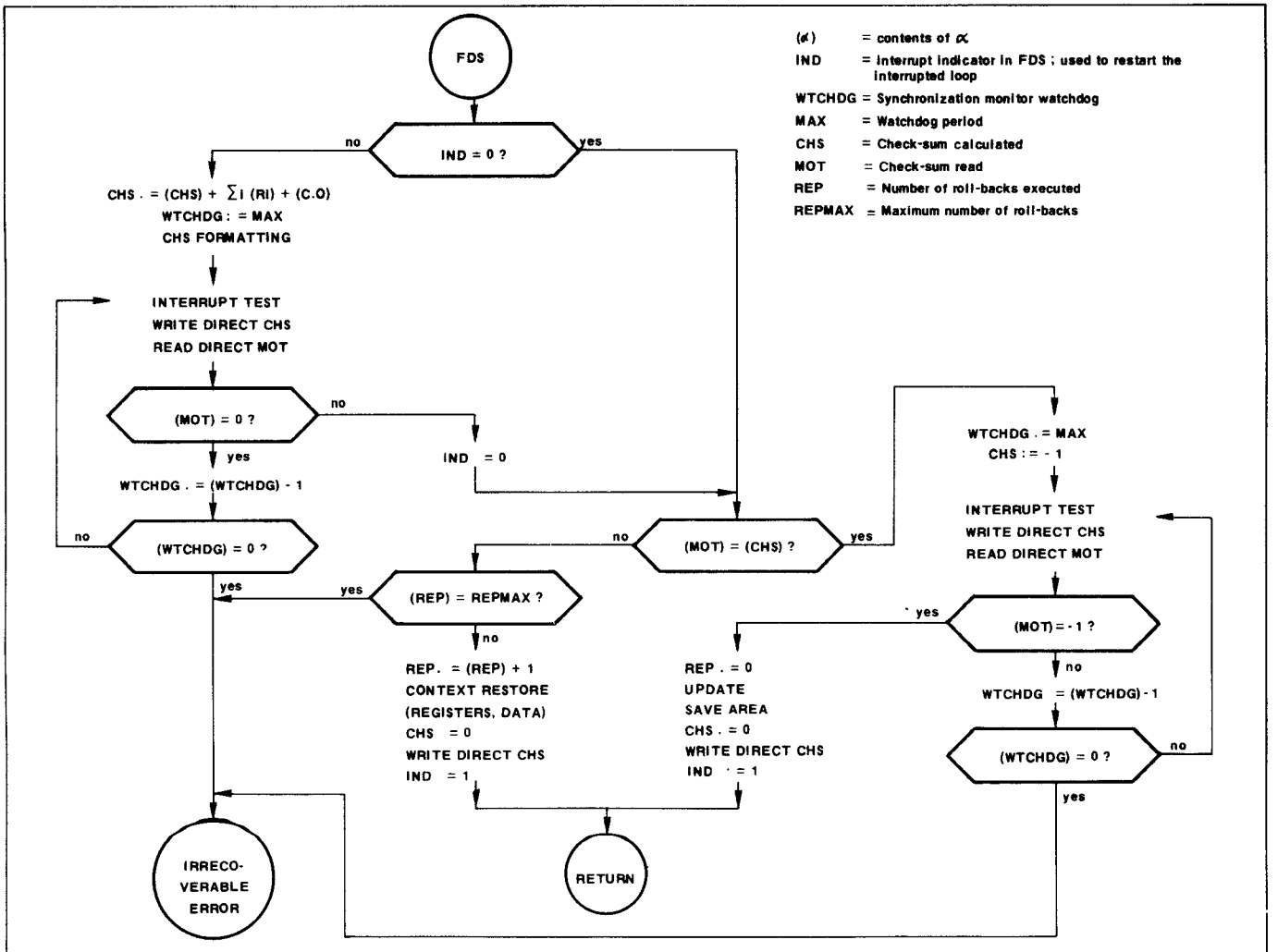

Fig 3 - "END OF SEQUENCE" MICROPROGRAM

Specific problems

Interrupt synchronization. To ensure correct bi-computer operation, it is necessary when the two computers exchange check-sums that they correspond to the same sequence. And since there is no microsynchronization between both computers, interrupts are not necessarily accepted at the same time by both computers.

To ensure "end of sequence" synchronization, it is sufficient :

1. That the interrupt acceptance microprogram first resets the MOT transmitted by the direct link.

2. That, for interrupts gathered on the same level, the reading of the status of one level by one of the computers isolates the level from the "outside" until the other computer reads the status.

Swapping problem. The Mitra 15 multi-tasking monitor (MMT) can request swapping on a system disk a task that has been interrupted and which therefore is not at the end of the sequence. Thus both copies of the task are different.

Two solutions are possible :

- have two system disks : one per computer ; the system disk is then a simple extension of the main memory ; this solution has the advantage of checking system disk operation and avoiding a too high synchronization of the computers during exchanges with the disk ;
- if only one system disk is available, roll-back on the task before swapping : both copies will then be in the same status.

Save stack size choice. Save stack size is a parameter which can be varied to obtain a speed memory occupation trade-off adapted to various applications.

To do this, the curve giving the ratio of the total number of "end of sequence" to the number of usable "end of sequence" (that is, that are not caused by a save stack saturation) as a function of stack size (fig.4) can, in particular, be studied for each type of application.

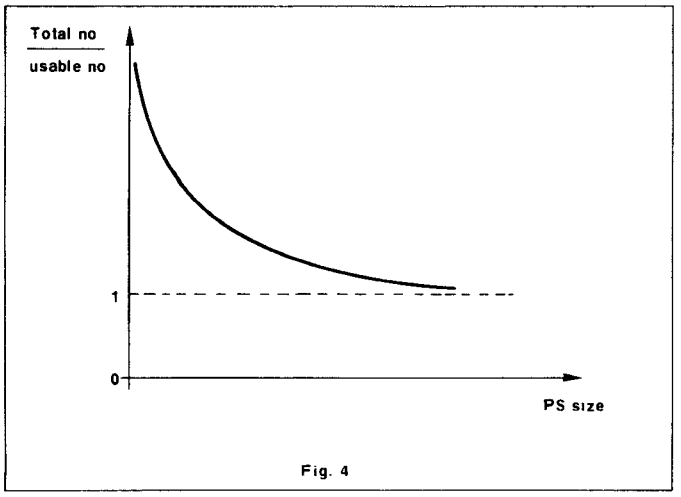

Fig. 4

The "optimum size" of the save stack will be selected versus the curve trace, memory size, and time constraints.

If due to safety reasons sequences in a very disturbed environment should be short to avoid too many intermittent failures which might cause irrecoverable errors, a save stack with a smaller size than that defined for the previous criteria can be selected.

Error procedures. If an irrecoverable processing error occurs, a proper operation test (3) will be initiated by program on each computer to designate the failed computer. In this case, the other computer is valid and may be used to continue processing (without redundancy or with temporal redundancy : chapter "Marignan Single-Processor : Temporal Redundancy")while the first is being repaired.

If the proper operation test (TBF) does not detect a failure, either because the failure is permanent but is not among the failures detected by TBF, or because the irrecoverable error was due to one or several intermittent failures, the operator will either have to halt processing or continue it on either one of the computers.

If an error is detected by the trap system or by the monitor credibility tests, the faulty sequence will have to be rerun to check if the failure is an intermittent or permanent failure ; in the latter, the current program has to be abandoned.

If an I/O error occurs (detected by the coupler : paragraph " Hardware structure"), the microprogram ensuring the transfer can, according to the type of peripheral, or the error detected, either retry the transfer, or continue processing on only one of the computers (arbitrarily chosen or not), or halt processing.

Efficiency and WLR

Efficiency. The efficiency in detecting errors due to intermittent failures or not in the doubled system(central processing unit, main memory, minibus direct link, coupler and system disk in case of dual disk system) is very near 100 %. On the other hand, these errors are detected before they could be propagated outside the task which ensures high system safety.

The majority of the errors due to intermittent failures will be corrected by the system. However, the following errors will not be corrected :

- errors due to permanent failures,
- stored data losses (for dynamic MOS memories badly refreshed).
- faulty rewritings after a sequence of readings (for destructible read memories),
- addressing errors only on a half-cycle (destructible read memories),
- addressing errors during saving phases causing data destruction,
- addressing errors contaminating data of a task initiated before the end of the current sequence,
- errors extending to both the processing and saving phases during the same sequence.

Except for the first two types of errors, these errors are "second order", that is, their probability is on the order of p^2, if p is the probability an error due to an intermittent failure occurs during a sequence.

They will be corrected only by auxiliary devices : memory protection, redundant codes on data and addressing, read and write circuitry test, etc... which permits designating the faulty memory and therefore continuing with the other computer or duplicating the disturbed areas before resuming with both computers if the failure was intermittent.

The efficiency in correcting errors due to intermittent failures can therefore be estimated to be greater than 90 %.

WLR. The hardware is almost 2.1 times the hardware of an equivalent non-redundant system.

Execution time is increased from 15 to 25 % depending on the save area size, the proportion of memory modification instructions, and the proportion of instructions causing an end of sequence.

WLR is thus approximately 40 %.

Marignan Single-Processor : Temporal Redundancy

For a permanent error, we saw (in paragraph "error procedures") that a proper operation test was initiated to designate the failed computer and to continue processing on the other computer throughout the duration of the repairs.

Since this repair may be rather long or impossible (systems onboard satellites), it is advisable to keep tolerating intermittent failures eventhough program execution might have to be slowed down. Therefore, we are going to attempt to apply the Marignan method to temporal redundancy.

The hardware structure is thus that of a single computer. The software structure includes a save area that shall contain two check-sums CHS (1) and CHS (2) instead of only one.

The operation is as follows : at the first end of sequence, the check-sum in CHS (1) is saved, a roll-back procedure is carried out and the sequence is re-executed. At the end of this second execution, the check-sums CHS (1) of the 1st execution and CHS (2) of the 2nd execution are compared.

If they are identical, the save area is reinitialized and program execution is continued. If they are different, the sequence is resumed and the last two check-sums are compared. Figure 5 shows the flow chart analyzing the running of a sequence.

This method when applied to a single processor has the following disadvantages :

- it does not detect permanent failures. Therefore whenever a sequence is terminated, a proper operation test (3) has to be initiated :
- an error in the save during the first execution of the sequence may cause an irrecoverable error in the following executions. This type of error although highly improbable may be unacceptable in certain applications.

To avoid this disadvantage, a second solution consists of doubling the data areas (and if necessary, the program area if no memory protection is available) and the save area. A roll-back is executed then only if the check-sums are different. This solution would be costly in memory space but less costly in execution time.

If the proper operation test is not considered, detection efficiency can be estimated at 80 % for the first solution (100 % - 15 % for permanent failures - 5 % for save errors) and 85 % for the second solution. The efficiency in correcting errors due to intermittent failures can be estimated at 85 % for the first solution and 90 % for the second.

The additional hardware with respect to a standard single processor is restricted to a main memory increased in capacity for the save areas (and, if necessary, data areas) and a longer control memory occupation.

Execution time is more than doubled.

Work-load ratio must therefore be around 0.4.

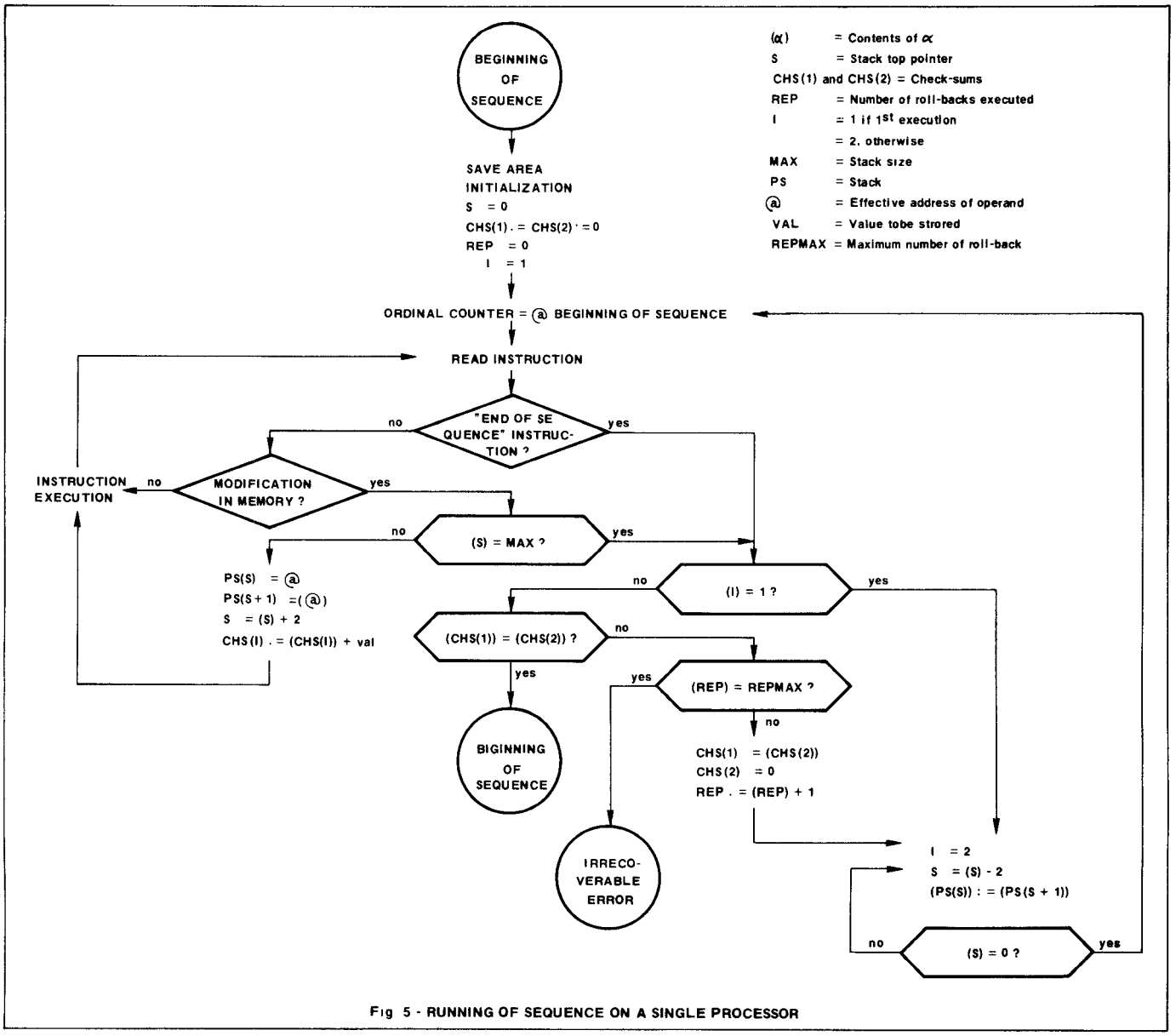

Fig 5 - RUNNING OF SEQUENCE ON A SINGLE PROCESSOR

Conclusion

The Marignan Method applied to a bi-computer has an efficiency close to 100 % for a work-load ratio around 0.4. In addition to tolerating intermittent failures, it permits effective detection of permanent failures. A reconfiguration in "degraded mode" permits tolerating one permanent failure.

Moreover, Marignan requires only slight modification of the standard hardware and standard software to which it is applied. In particular, the Marignan structure is transparent to the user. Production and operating costs are thus very low for a very high-reliability system intended for small-scale manufacturing.

In addition, Marignan seems applicable to any system regardless of its size or structure.

Applications to microprocessors, multiprocessors, and multicomputers are being studied. Experimenting is planned to be shortly carried out on two Mitra 15's.

(1) " Effects and detection of intermittent failures in digital systems".
M. Ball, F. Hardie - AFIPS, FJCC 1969, pp 329 - 335

(2) " Rhe retryable processor".
G.H. Maestri - AFIPS, FJCC 1972, T1, pp 273 - 277

(3) G. DEBROCK
Rapport faisant partie du 2ème lot du contrat DRME 73/375. 31 Janvier 1974.

(4) " Détection et correction des erreurs dues aux pannes fugitives ".
Y. DESWARTE
Rapport faisant partie du 2ème lot du contrat DRME 73/375. 31 Janvier 1974.