General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

NAG1-296 DAA

SYNCHRONIZATION AND FAULT-MASKING IN REDUNDANT REAL-TIME SYSTEMS[1]

C. M. Krishna, Kang G. Shin

Computing Research Laboratory
Department of Electrical and Computer Engineering
The University of Michigan
Ann Arbor, Michigan 48109
(313) 763-0391

and

Ricky W. Butler

NASA Langley Research Center
Hampton, VA 23665
(804) 865-3681

RECEIVED NASA STI FACILITY ACCESS DEPT.

ABSTRACT

A real-time computer may fail because of (i) massive component failures or (ii) not responding quickly enough to satisfy real-time requirements. An increase in redundancy -- a conventional means of improving reliability -- can improve the former but can -- in some cases -- degrade the latter considerably due to the overhead associated with redundancy management, namely the time delay resulting from synchronization and voting/interactive consistency techniques.

In this paper, we consider the implications of synchronization and voting/interactive consistency algorithms in N-modular clusters on reliability. All these studies have been carried out in the context of real-time applications. As a demonstrative example, we have analyzed results from experiments conducted at the NASA Airlab on the Software Implemented Fault-Tolerance (SIFT) computer. This analysis has indeed indicated that in most real-time applications, it is better to employ hardware synchronization instead of software synchronization, and not allow reconfiguration.

Subject Index: Synchronization, Communication (1)

[1]The work reported in this paper was supported in part by NASA Grant No. 1-296. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of NASA. All correspondence regarding this paper should be addressed to Professor Kang G. Shin.

(NASA-TM-87478) SYNCHRONIZATION AND FAULT-MASKING IN REDUNDANT REAL-TIME SYSTEMS (NASA) 34 p HC A03/MF A01 CSCL 09B

N85-28644

Unclas
G3/62 23763

1. INTRODUCTION

The use of digital computers, particularly multiprocessors, has become commonplace in such real-time applications as aircraft control, nuclear reactor control, power distribution and monitoring, automated manufacturing, etc. Such computers are typically required to have very high reliability. (For example, the benchmark figure used at NASA is 10^{-9} probability of failure over a 10-hour flight period for an aircraft control computer). Unlike their conventional counterparts, the failure probability of real-time systems is not completely characterized by the probability of massive hardware failure alone : failure can also occur due to excessively long response times. In other words, there are *hard deadlines* for code execution that, if missed, can lead to catastrophic consequences. The *probability of dynamic failure*, p_{dyn}, introduced in [1] and further refined in [5], integrates the two causes of failure and expresses their probability of occurrence. p_{dyn} over some pre-defined operating period is defined as the probability that the system will miss at least one hard deadline during this period.

As a solution to the reliability problem in real-time computers, N-Modular Redundancy (NMR) has been widely used. In such cases, a task is executed in parallel by N modules, and it is necessary to reach agreement even in the presence of faults among some of these N modules. The traditional method to do this is by voting. However, such approaches are not guaranteed to work when it is required to unambiguously locate a fault as well as to detect the consequent error. Indeed, simplistic analyses (e.g. [18,19]) are negated by the possibility that a malfunctioning device will act *maliciously* or disruptively. A faulty device can do this by reporting different values to different receivers, i.e. by *lying*. For this purpose *Byzantine Generals* (also known as *interactive consistency*) approaches [2-4] have been suggested. However, such algorithms have very high over-

heads, and there are cases where Byzantine General algorithms actually make a multiprocessor more liable to failure by *adding* redundancy beyond a certain threshold. That is, by adding redundancy it is possible to introduce too much overhead to meet the p_{dyn} requirement. There is therefore the need to examine carefully the implications of various methods for handling NMR systems. In this paper, we critically compare several approaches in the context of real-time applications.

The block diagram in Figure 1 shows a typical real-time system from an functional point of view. Sensor input is treated by means of voting or interactive consistency methods to ensure fault-tolerance, before being made available for execution by the NMR cluster. Once execution is complete, voting or interactive consistency checks have again to be performed. With that done, the output data can be passed on to the actuator(s) or the displays. Throughout, synchrony must be maintained amongst the processors that make up the NMR cluster.

Response time is defined as the delay between the triggering of a job by the arrival of the relevant sensor input to the actuator/display output that finally results. The overall response time is the sum of the time for reaching agreement on the sensor data, the execution time itself, and the time for synchronization and reaching agreement on the results generated by the processors in the NMR cluster before the data are fed to the relevant actuators. Of these times, only the execution time is independent of the cluster size, N.

In this paper, we present the results of a study of the delay overheads imposed by voting/interactive consistency and synchronization methods and quantify their impact on computer reliability expressed through the probability of dynamic failure. This is the probability that, over a certain given period of operation, either (a) the processor cluster

has failed to the point that fault-masking no longer works, or (b) the hard deadline has been violated although (a) has not occurred. In particular, we shall contrast the reliabilities offered by reconfigurable and non-reconfigurable systems. It will turn out in some cases, contrary to intuition, that non-reconfigurable systems are the more reliable type.

This paper is organized as follows. In the next section, we consider synchronization and the delay it imposes. In doing so, we present a theorem that enables the design of an arbitrarily large phase-locked, fault-tolerant clock. We show that software synchronization techniques are excessively time-consuming, and indeed impose a limit on the size of a cluster that can be thus synchronized. In Section 3, the fault-masking techniques of voting and interactive consistency are considered, and their delay overheads estimated. In Section 4, we use the results of Section 3 to compare the reliabilities of reconfigurable and non-reconfigurable systems, operating in real-time, under the constraints of a hard deadline. We conclude with Section 5 which identifies a number of interesting unsolved problems.

2. SYNCHRONIZATION

When a multiplicity of processors executes code in parallel, care must be taken to keep them reasonably in step. Therefore, the issue of synchronization is focal to all methods of forward error recovery. There are two basic methods of synchronization:

(1). Each processor has an ultra-precise clock. When the computer is switched on, the clocks are synchronized. If the clocks are sufficiently precise, the processors will continue to run in lock-step for an appreciable period. Unfortunately, such highly precise clocks are extremely expensive, and unsuited to incorporation in computer circuits. (For a description of highly precise clocks, see [7]). Clocks used in computer circuits drift too rapidly for this method to be employed in practice. We shall

not consider this method any further.

(2). The synchronization is carried out mutually. There is no single component whose functioning is critical to the security of the whole system. One may choose to synchronize either the individual processor clocks or the processors at pre-defined boundaries of software execution. In the first case, one has a system operating more or less in lock-step, such as the FTMP system [8]. Both methods of synchronization are based on the same basic concepts; the only difference is the frequency with which synchronization is carried out.[2] The notion of *virtual time sources* now arises naturally. These are not necessarily clocks in the traditional sense; they mark the points at which an individual processor performs synchronization. It is convenient to view them as *virtual clocks,* whose transitions represent either clock "ticks" or execution of a stretch of code up to a pre-specified boundary. In the sequel, unless it is otherwise stated, the term "clock" is used to mean "virtual clock".

When synchronization is mutual, no "absolute" underlying time-source exists, only a set of time-sources whose relative behavior must be kept in step. The synchronizer (which may or may not be physically part of the processor and which may be implemented either in hardware or in software) must therefore in each case have a perception of the state of the other time-sources. This perception may or may not be identical to that of the other synchronizers: if faulty modules necessarily behave consistently with respect to all synchronizers, it is identical; otherwise it need not be so.

[2] An important corollary of this is that the maximum clock drift rates permitted in the synchronization at boundaries of a large stretch of code are smaller.

The synchronization process contributes to the system overhead in two ways. Firstly, there is the overhead imposed by the synchronization task itself. Secondly, the task-to-task communication overhead is proportional to the degree of synchronization achieved. If hardware synchronization with phase-locked clocks is employed, the synchronization overhead can be reduced to vanishing point. If software synchronization is used, the overhead is significant. Both approaches to synchronization will be considered in succeeding sections. First, however, we consider the impact on the communication overhead.

Because of severe timing constraints, real-time systems do not generally use sophisticated mechanisms for task-to-task communication. Typically, data are transmitted from one task to another via timing rules agreed in advance. As a result, the receiving task has to wait for a time equal to the sum of the maximum transmission time and the maximum possible clock skew before it can read the data. For example, in the SIFT system, a wait loop has to be added when the normal dispatching overhead is less than the maximum clock skew. Such a wait naturally degrades performance, and all the more so since execution is supposed to be carried out in real-time. Where synchronization is carried out in software and depends on the transmission of timing data on regular data channels, this transmission delay feeds back to increase the synchronization delay itself. We will consider this matter in detail in Section 2.2.

2.1. Hardware Synchronization

Probably the best way of keeping clocks synchronized in the presence of malicious failures is to lock them in phase. For an introduction, see [12], and for a careful analysis, see [14].

The philosophy behind the operation of phase-locked clocks is simple. See Figure 2. Given N clocks to be synchronized in the face of up to m faulty clocks, the arrangement consists of a receiver which monitors the clock pulses of the N–1 other clocks in the arrangement, and uses them as a reference. This reference is treated as a measure of the phase error by the receiving clock. The estimated phase error is then put into an appropriate filter, and the output of the filter is used to control the clock oscillator's frequency. By appropriately controlling the frequency of the individual clocks, they can be kept in phase-lock and therefore in synchrony.

For the arrangement to work properly, the phase error should be below some critical value. The magnitude of this value depends to a large extent on the nature of the filter. A brief discussion of clock stability is provided in [12]. The arrangement for N=4, m=1, is particularly simple and is, to our knowledge, the only phase-locked clock constructed. Here, the reference used is the second incoming pulse (in temporal order), i.e. the median pulse. Such a clock is proof against the malice of a single faulty clock.

Unfortunately, when one attempts to increase N or m without sufficient care, synchrony can be lost through malicious behavior and the consequent formation of a multiplicity of *cliques*, i.e. self-sustaining groups of clocks maintaining synchrony within their individual cliques, but out of synchrony with the other non-faulty clocks in the cluster. By definition, there must be one and only one clique to which all non-faulty clocks belong. In this section, we show how phase-locked clocks may be safely expanded to tolerate *any* given number of malicious clocks. In particular, we seek to determine N as a function of m, and also the voting strategy.

A simple voting strategy is defined by an integer $f(N)$, which means that the $f(N)$-*th* incoming signal (in temporal order) is taken as the reference value against which

the receiving clock compares its own value. Our task is so to obtain N and $f(N)$ that there is, in the face of up to m malicious clocks, exactly one clique in the whole system to which every non-faulty clock belongs.

To begin with, note that as long as synchrony is maintained, every non-faulty clock must belong to exactly one clique, while a maliciously faulty clock may belong to any number of cliques. Using this requirement, we can obtain the following useful theorem.

Theorem 1: Let N and m be the cluster size and the maximum number of faulty members tolerable in the system, respectively, then (i) $N > 3m$ and (ii) $f(N)$ is the smallest integer strictly greater than $\frac{1}{2}(N+m) - 1$.

Proof: For convenience, define the *size* of any clique as the number of non-faulty clocks in it. Then, the minimum possible size of a clique is $f(N)-m+1$. There are at least $N-m$ non-faulty devices. The requirement that there be exactly one clique of non-faulty devices then yields:[3]

$$\lfloor \frac{N-m}{f(N)-m+1} \rfloor = 1 \qquad (1)$$

Also, since the $f(N)$-*th* incoming signal must either be a signal from a non-faulty clock or from a faulty clock that is sandwiched between the signals of two non-faulty clocks, we have $f(N) \leq N-m-1$. From (1) we get $f(N) > \frac{1}{2}(N+m) - 1$, thereby leading to an inequality for $f(N)$:

[3] It is easy to show that if this expression is satisfied, there will be exactly one clique when the number of maliciously malfunctioning clocks is less than m.

$$\frac{1}{2}(N+m)-1 < f(N) \leq N-m-1 \qquad (2)$$

This inequality implies $N > 3m$. We may therefore define $f(N)$ to be the smallest integer strictly greater than $\frac{1}{2}(N+m) - 1$ where $N > 3m$. Q.E.D.

With this arrangement, the possibility of forming multiple cliques vanishes, and the system remains in synchrony as long as the individual non-faulty clocks have sufficiently low drift rates and are sufficiently well synchronized at start-up to maintain phase-lock. As an illustration of the above theorem, consider the following example.

Example: This example is meant to illustrate why $N > 3m$ in a phase-locked clock.

Case 1: First, consider the case N=6, m=2, and simple majority voting is used. Label the clocks A, B, C, D, E, and F respectively. Let E and F be maliciously faulty clocks. We present here a scenario in which cliques {A,B,E,F} and {C,D,E,F} are formed, thus destroying synchrony between, for example, B and C.

Let $C_\alpha^{(i)}$ denote the moment in "absolute time"[4] at which clock α for α=A,B,C,D,E,F puts out its *i-th* tick. Since even non-faulty clocks drift, these moments will very likely be mutually different. Define the α-scenario, $S_\alpha^{(i)}$ as the ordering of the clock values as seen by clock α in the *i-th* clock cycle. In the sequel, we suppress the superscript for convenience. Denote the reference clock by bold-face notation. The following scenarios are possible for, say, the *i-th* clock cycle:

$$S_A = C_A < C_E < C_F < C_B < C_D < C_C$$

[4] While it might seem a contradiction to invoke real or absolute time when we said that it has no meaning in mutual synchronization, it is not so. The only role "absolute time" has to play here is in defining a mutual ordering of the respective clock ticks.

$S_B = C_E < C_F < C_A < C_B < C_D < C_C$

$S_C = C_A < C_B < C_D < C_C < C_E < C_F$

$S_D = C_A < C_B < C_D < C_C < C_E < C_F$

and the scenarios S_E and S_F are of no relevance since E and F are faulty clocks. Note that the mutual order of the non-faulty clocks is preserved for all four scenarios, while that including the faulty clocks is not necessarily preserved.

It is easy now to see that this scenario can perpetuate itself and that the sets {A,B} and {C,D} will then diverge, forming cliques {A,B,E,F} and {C,D,E,F}, each of size two. Phase-locking is no protection against this because of the malicious behavior of faulty clocks E and F. Therefore, N=6 is not large enough to tolerate up to m=2 faults.

Case 2: Now consider N=7, m=2, and $f(N)$=4, with non-faulty clocks A,B,C,D, and G, and faulty clocks E and F. We show that multiple cliques cannot form by systematically eliminating all possibilities. Consider all possible 2-clique arrangements: {1,4}, and {2,3} where the numbers denote the size of the candidate clique. It is easy to see that a clique of size one is not self-sustaining since even with the two faulty clocks thrown in, there would only be two signals into the receiver circuit, which is less than $f(N)$. The {2,3} arrangement can similarly be disposed off: while the clique of size 3 is self-sustaining, that with size 2 is not. So, there is no two-clique arrangement that can be sustained. It is easy to see that three-, four- and five-clique arrangements are also impossible. So, N=7, m=2, and $f(N)$=4 has at most one clique.

Remark 1: A pathological curiosity

While the system is in phase-lock, it is possible for only $x=f(N)-m+1$ non-faulty processors to cause the whole system to drift monotonically in frequency. This can only

happen, however, when x of the processors have a consistent drift in one direction. Theoretically, the whole system could be dragged down to zero frequency or up to a very high frequency, keeping perfect synchrony as it gets there. Since the drift of good crystal clocks is only some few parts per million and the frequencies over which they can range is limited, this difficulty is no more than academic. It does, however, serve to emphasize that phase-locked clocks do not necessarily average out drifts.

Remark 2: Synchronization overhead

In the case of a phase-locked clock, there is some time overhead due to the oscillations that are possible as a result of malicious behavior. However, these are minimal when good crystal clocks are used, and so it is reasonable to treat the overhead of hardware synchronization as negligible. Also, the clock skew is very small/negligible in a well-designed phase-locked clock.

Remark 3: An alternative design

The only other hardware arrangement that we are aware of for keeping synchrony in the face of malicious behavior is the multi-stage synchronizer arrangement proposed by Davies and Wakerly [6]. The idea is shown in Figure 3. It consists of m stages of N synchronizers each. The system works on the principle that with this redundancy, there must be at least one level of synchronizers that assures proper synchronization in the presence of malicious faults. An informal proof is provided in [6].

This arrangement results in a proliferation of hardware. As may readily be verified, the total number of devices (processors and synchronizers) in the cluster is $2m^2+3m+1$. The total number of I/O ports required is given by $8m^3+10m^2+10m+2$. The potential enor-

mity of the above numbers should be driven home by the consideration that in order to maximize returns from redundancy, the individual modules must be isolated from one another as much as possible. This dictates that power supplies must also be replicated in large numbers, and that the benefits of large-scale integration cannot be brought to bear on the issue: individual synchronizers must be on separate devices -- even, perhaps, on separate cards. Otherwise, correlated and common-cause failures could wipe out reliability gains made by device redundancy.

Compared with the gargantuan redundancy required by the Davies and Wakerly approach, the $N=3m+1$ requirement of phase-locked clocks represents an extremely elegant hardware solution to the problem of synchronization in the presence of malicious faults.

2.2. Software Synchronization

The use of decentralized algorithms for synchronization offers an alternative to the hardware methods described above. Such algorithms enable a system consisting of many processors with their own clocks to operate in close synchrony. The degree of synchronization obtained by these algorithms depends primarily on the performance of the communications system, the precision of the clocks, and the frequency of resynchronization. The task-to-task communications system's one-way message time is at least $B+\delta$ where B is the maximum transmission time and δ is the maximum clock skew. The most time-efficient of the software synchronization algorithms that we know of is the *interactive convergence* algorithm [9].

In the interactive convergence algorithm, each processor in the system determines its skew relative to every other processor in the system. If any relative skew is greater

than a predetermined threshold, it is set to zero. An average of all the relative skews is calculated and used to correct its clock.

The following theorem (a trivial adaptation of one proved in [0]) characterizes the maximum clock skew of the system in terms of the system parameters defined below:

ϵ - maximum error in reading another processor's clock

ρ - maximum drift rate between any two clocks in the system

N - number of clocks in the system.

m - maximum number of faulty clocks accommodated.

R - resynchronization period.

$S(N)$ - execution time of the resynchronization task for an NMR cluster.

δ_0 - maximum clock skew at start-up.

Theorem 2: Let the following conditions hold:

$$3m < N$$

$$\delta \geq [1-\frac{3m}{N}-2\rho(1-\frac{m}{N})]^{-1}[2\epsilon\{1+\rho(1-\frac{m}{N})\} + \rho\{R + 2\frac{N-m}{N} S(N)\}]$$

$$\delta \geq \delta_0 + \rho R$$

$$\max(\delta, S(N)) < R$$

$$\rho\delta << \epsilon$$

Then, the non-faulty clocks remain in synchrony, i.e. the maximum skew is δ.

The synchronization algorithm is run periodically, the major component of the execution time usually being the time required to read every other processor's clock in the system. In the SIFT system, each processor's clock value is broadcast during a window of time allocated to it. There are N such windows, one for each processor in the system.

All other processors wait during this window to receive the broadcast data value.

In order to accommodate the worst-case situation, each window must be at least $B+\delta$ long. The interactive convergence algorithm takes an execution time equal to $S(N)=N(B+\delta)+K$, where K is the time needed to compute and carry out the clock correction.

It should be noted that this execution time of the synchronization task affects the synchronization process itself. Indeed, since this is a function of N, there is a maximum cluster size that can be synchronized in this way. To see this, substitute the above expression for $S(N)$ in the formula for δ, and obtain:

$$\delta \geq N[N-3m-2\rho(N^2+N-mN-m)]^{-1}\ [2\epsilon+\rho\{R+2(N-m)(B+\frac{K}{N})\}] \qquad (3)$$

From this, one can (a) compute the minimum execution time of the synchronization task as a function of the cluster size, (b) obtain the quality of synchronization (the smaller the δ, the better the synchronization), and (c) determine the largest possible cluster that can be thus synchronized: this is the largest N for which $S(N) < R$.

The values for the SIFT system are given by B=18.2 micro-seconds, and execution time for the synchronization task is 1.760 milli-seconds. Numerical results on the synchronization overhead using these values are plotted in Figure 4. The curves in Figure 4 show the dependence of the synchronization overhead on the maximum drift rate, ρ. The maximum cluster size permissible for synchronization is tabulated in Table 1.

Although the expression $S(N) = N(\delta+B)+K$ was presented as emanating from the SIFT system, it is easy to see that in *any* system where communication is by broadcast, and clock transmission slots are pre-determined, this expression will hold. It should also be reiterated that such communication protocols are the most commonly used protocols

in real-time systems. In any case, it is obvious that whatever the protocol used, $S(N)$ is very unlikely to be less than a first order function of N.

Even if, in a hypothetical case, $S(N)$ were negligible (which, of course can never happen but nevertheless represents an extreme case), δ will continue to be a function of N, and there will be a point for which $\delta > R$, at which synchrony will break down.

3. VOTING AND BYZANTINE GENERALS ALGORITHM

A complete analysis of real-time systems must take into account both the processor synchronization overhead and the overhead associated with voting/interactive consistency techniques used for input and output functions. The former has been carefully examined in the previous section. We now discuss the latter in detail.

3.1. Voting

Once the delay involved in synchronization is taken account of, there is very little additional delay if the voting is carried out in hardware. With software voting, however, the additional overhead can be significant.

Voting in software is carried out by individual processors placing data to be voted on in pre-specified "mailboxes" or "pigeonholes". The voter searches the mailboxes for valid data, fetches them, and then votes on them. The execution time in the retrieval step is directly proportional to the number of processors in the cluster, N. The execution time required to vote N values and diagnose up to m faults is known to be at least $(N-1)C_1 + C_2$ but less than $[(N-1) + 2m - 2]\ C_1 + C_2 = [\frac{5(N-1)}{3} - 2]\ C_1 + C_2$ where C_1, C_2 are some constants (see [13] for this).

Experimental data exist for 3-MR and 5-MR in SIFT. These data can easily be introduced into the linear model obtained above. If s is the number of data values

voted, and $V_N(s)$ the time taken for an N-way vote on s data values, the following expression was found to hold for SIFT:

$$V_N(s) = 58.5\ s\ N + 129.5 \qquad (4)$$

where the time unit is micro-seconds. This is a large overhead: in SIFT, voting is performed at the beginning of a 3.2 milli-second subframe. If s=6, N=5, then 78% of the subframe is consumed by the voting algorithm [16].

3.2. Byzantine Generals Algorithms

The Byzantine Generals, or interactive consistency, algorithm must be used when it is needed to isolate the sources of errors as well as to mask the errors themselves. It finds use when reconfiguration upon failure is to be attempted and the executive is distributed. The algorithm takes into account the fact that faulty processors may be malicious, in other words, that they need not fail only in "safe" directions. To be absolutely certain that faulty processors can be properly identified for isolation, it is necessary to allow for every possible misbehavior. Thus the case when a faulty processor is malicious -- that it actively and intelligently attempts to hide its malfunction -- must also be handled. Such algorithms are typically used to reach agreement between processors in a cluster on incoming sensor data, and in certain clock synchronization algorithms. For further details, see [2-4].

The input of data is accomplished by every processor reading the external sources independently or by one processor reading the external sources and then distributing the obtained value to the rest of the processors. In the first case, each processor would very probably get a different value -- even if they were in perfect synchrony -- due to the inherent instability in reading analog data. Hence, a subsequent exchange of values read along with a mid-value selection is required to get a consistent value. However, this

process suffers from sensitivity to malicious faulty processors and *interactive consistency* (or Byzantine Generals) algorithms are essential where fault isolation and reconfiguration are required.

The interactive consistency algorithm consists of the following steps:

(1) The source value is distributed to the N processors.

(2) The received values are exchanged m times to handle up to m faulty processors.

(3) A consistent value is obtained by use of a recursive algorithm. When m=1, this reduces to a majority calculation.

N must be at least $3m+1$. The overhead for these interactive consistency algorithms can be considerable. The number of messages required to obtain interactive consistency is of the order of N^{m-1}. To give an idea of the actual numbers incurred in practice, some experiments on the SIFT computer were run.

In SIFT, with five-way voting, only one fault can be located. The simple flight-control applications currently running in SIFT use 63 external sensor values, each of which goes through the interactive consistency algorithm. From the data collected, execution times for steps (1) and (2) of the algorithm can be estimated, and a lower bound determined for step (3). The following data were measured: step (1) : 3.05 *ms*, step (2) : 2.22 *ms*, and step (3) : 6.57 *ms* (total 11.84 *ms*). For larger m, the step (1) execution time should not change significantly, while the step (3) calculation would require at least 6.57 *ms* (very likely much more). The step (2) process consists of only message exchanges and thus varies directly with the number of messages which are sent. The following formula represents an approximate execution time for step (2) as a function of m: 2.22 N^{m-1} *ms*). We may add the timing values for steps (1) and steps (3) above to this expression to obtain a lower bound for the overhead of the Byzantine Generals algorithm

in SIFT. Since the interactive consistency tasks must be executed at the data sample rate, a large portion of the available CPU time is consumed. In Table 2 we present the interactive consistency overhead in SIFT. It indicates that the percentage overhead drastically increases as (i) the number of faults to be tolerated increases, or (ii) the major frame (i.e. data sampling period) decreases.

It should be pointed out that there have lately been some more efficient implementations of the Byzantine Generals algorithm [18] than have been implemented on SIFT. However, even such implementations exhibit high overheads as the number of faulty modules to be accommodated increases.

4. RECONFIGURABLE AND NON-RECONFIGURABLE SYSTEMS

In some cases, it is possible through encryption, to defeat malicious failures, thus making interactive consistency algorithms unnecessary for locating faults. Unfortunately, we know of no estimate of the probability of occurrence of malicious faults that cannot be so defeated. In the absence of any such data, we must for safety, allow all faults to be so malicious that encryption cannot defeat them. This means that reconfigurable systems in ultra-reliable applications must use the interactive consistency algorithms to isolate faulty units.

Unfortunately, as we have seen, this algorithm is extremely time-consuming to run. Reconfigurable systems must therefore contend with a large overhead as compared to non-reconfigurable systems.

However, reconfigurable systems have the advantage of dynamic redundancy management. When widespread failures occur, it is possible to retire some clusters in order to keep others at full strength. Also, by periodically purging itself of faulty components, a reconfigurable system can survive in the face of more failures than can a

non-reconfigurable system. For example, if one started operation with a 7-cluster, the reconfigurable system would not fail unless either (a) all but two processors fail, or (b) more than m (m=2 for a 7-cluster, and 1 for a 4-cluster) processors fail between successive tests, while the corresponding non-reconfigurable system would fail if more than 3 processors failed. This does not mean that a reconfigurable system is necessarily better than a non-reconfigurable one, since as we shall see, timing requirements impose severe constraints on the size of reconfigurable clusters.

We shall contrast the reliability of reconfigurable and non-reconfigurable systems with the following example. Assume that there is a single critical task in the system that requires 3.2 milli-seconds to run, and that this task is dispatched every 100 milli-seconds.[5] The hard deadline is violated when a task is not completed before its successor is dispatched. There is a total of N processors available. Processors fail according to an exponential law with specified MTBF. The mission lifetime (duration between successive service stages) is also specified.

Remark 4: It is usual in analyses of multiprocessors to assume that the service time is exponentially distributed. This works well for general-purpose systems where it represents a certain lack of information about the job mix, and where the long tail of the exponential distribution does not seriously affect computations since these are generally based on mean-value analysis and do involve hard deadlines. Real-time software, however, is written according to strict timing requirements, and one must have a good idea about the maximum execution time. One may choose to regard this maximum time as the actual execution time in all instances for purposes of safety. The value of 3.2 milli-seconds that we adopt here is the length of a single subframe in the SIFT system.

[5] 3.2 and 100 milli-seconds represent, respectively, the length of a subframe and frame in the SIFT system.

In the following sections, we consider non-reconfigurable and reconfigurable systems separately. In both cases, we assume that synchronization is by means of phase-locked (physical, not virtual) clocks. Since these can be made arbitrarily reliable and are common to both reconfigurable and non-reconfigurable systems, we do not consider the probability of clock failure in what follows. Numerical results in the section on reconfigurable systems are based on the lower bounds obtained from SIFT.

4.1. Non-Reconfigurable System

Under the above assumptions, the probability of dynamic failure is simply equal to the probability of static failure, i.e. the probability that fewer than $\lceil \frac{N}{2} \rceil$ processors fail over the mission lifetime. This probability for a number of mission lifetimes is graphed in Figure 5.

4.2. Reconfigurable System

Since, in a reconfigurable system, an execution of the Byzantine Generals Algorithm must take place both at the input and the output, we have the time overhead bounded below by $4.44N^{m-1}+19.24$ milli-seconds ($N > 3m$). Since the task execution time is 3.2 milli-seconds, an unreplenishable reserve of 100–3.2=96.8 milli-seconds of time is available. If the overhead is smaller than this, the probability of dynamic failure is equal to the probability of static failure, if not, it is equal to unity.

As may readily be seen, for clusters with $m>2$, the overhead exceeds the reserve of time, so that the maximum allowed size of the cluster is N=7, m=2. For this reason, if we start with more than 7 processors, the excess processors will have to be on stand-by for inclusion upon a failure within the cluster. Failure occurs if either during a single

execution more than m failures occur in the cluster, or if the processor pool is exhausted, i.e. if there is an insufficient number of processors left to make up a cluster.

The reconfiguration policy is simple. The system begins operation with either a 7-MR or a 4-MR cluster (depending on the value of N). As processors fail, they are replaced if spares are available. If the stock of spares is exhausted, further failures are handled by the 7-MR cluster reconfiguring into a 4-MR cluster (when the system began operation with a 7-MR cluster). If $N < 7$, only a single failure can be tolerated. It is thus a combination of hybrid and adaptive voting [17].

The analysis for the dynamic failure probability is straightforward, and differs from previous analyses on reconfigurable NMR [18,19] in that the probability of more than m failures over an execution cycle is taken into account here. We assume that the failure rate of the stand-by processors is the same as that of functioning processors.

Numerical results for the probability of dynamic failure of reconfigurable systems are plotted in Figure 5 for a ready comparison with their non-reconfigurable counterparts.

It is apparent from Figure 5 that while increasing the number of available processors in a non-reconfigurable system reduces its probability of dynamic failure, there is a lower bound to the probability of dynamic failure for reconfigurable systems. This bound is caused by the fact that the cluster size is limited to 7, since the overhead exceeds 100% for larger clusters. There is therefore a point after which the probability of more than m processor failures over a single execution (aggregated over the mission lifetime) becomes the dominant component in the probability of dynamic failure. As one might expect, the reconfigurable system performs better than the non-reconfigurable system when the mission lifetime is longer. This has been at the horrible price, in this case, of a

50.3% overhead.

Clearly, the results in Figure 5 are problem-specific, indeed, they are critically dependent on the length of the inter-dispatch interval, the unreplenishable reserve of time that is available, and the time taken to execute the Byzantine algorithm. Also, while the reconfigurable system may appear to be the better performer in Figure 5, this is largely due to the large reserve of time available. Suppose that the task, instead of taking a maximum of 3.2 milli-seconds to perform, took 70 milli-seconds. Then, the reserve of time is 100–70=30 milli-seconds, and the largest reconfigurable cluster that could fit in this reserve is $N=4$, $m=1$. In Figure 6, we display results for such a task. Naturally, the reconfigurable system comes out much more poorly here.

In place of running the Byzantine Generals Algorithm every time the task is dispatched, one might consider running it only when execution of a vote detected a fault. While this would reduce the expected overhead, the worst-case overhead would still be the same. For this reason, while such a scheme might reduce the expected critical task execution time, and allow more (non-critical) tasks to be run, the maximum number of *critical* tasks possible is not increased.

However, if one were interested in expected response time as a secondary attribute of system performance, this scheme is worth exploring. In particular, such a scheme would reduce the *mean cost* as defined in [1].

5. CONCLUSIONS

In this paper, we have considered various methods of synchronization and fault-masking in redundant systems, with a view to using them in real-time applications.

We have presented a theorem that makes it possible to design fault-tolerant phase-locked clocks of arbitrary size, thus making hardware synchronization of arbitrarily large

clusters possible; quantified the overheads due to software synchronization, showing that these impose a large overhead, and that they break down beyond a certain cluster size. We have presented experimental results obtained from the SIFT computer on the overheads connected with software synchronization, software voting and interactive consistency algorithms. Finally, we have contrasted the reliabilities of reconfigurable and non-reconfigurable systems, under conditions of hard deadlines and unreplenishable reserves of time, and shown that there are conditions under which the large overhead connected with reconfigurable systems actually reduce their reliability below that of their non-reconfigurable counter-parts.

A number of interesting and important questions remain open. We list a few of them below.

(i) How can one definitively estimate or measure the probability that a processor fault will act maliciously?

(ii) What effect will various coding algorithms have on the probability of malicious failure?

(iii) The quantity $3m+1$ occurs repeatedly for the minimum number of elements in a cluster that must maintain either synchrony or interactive consistency in the face of up to m malicious failures: in this paper, we have added phase-locked clocks to this family of structures. This would seem to be caused by some underlying common features or principle. As of this writing it is unknown, as to whether or not this is so, and if it is, what the underlying principle is. The answer to this question would very considerably add to our understanding of the behavior of distributed systems.

The results and indications presented in this paper are useful in the quest for appropriate modeling and analysis of real-time systems. Real-time systems have too often been treated as an unimportant appendage of general multiprocessing. It is important to recognize that real-time systems form a separate discipline of their own, and to develop tools for their development. The work outlined in this paper forms part of such an effort.

REFERENCES

[1] C. M. Krishna and K. G. Shin, "Performance Measures for Multiprocessor Controllers", *Performance '83,* edited by A. K. Agrawala and S. K. Tripathi, North Holland, pp. 220-250, 1983.

[2] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM,* Vol. 27, No. 2, pp. 228-234, April 1980.

[3] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Prog. Lang. and Syst.,* Vol. 4, No. 3, pp. 382-401, July 1982.

[4] D. Dolev, "The Byzantine Generals Strike Again," *J. Algorithms,* Vol. 3, No. 1, pp. 14-30, January 1980.

[5] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "The Application to the Aircraft Landing Problem of a Unified Method for Characterizing Real-Time Systems," *Proc. IEEE Real-Time Systems Symp.* Arlington, VA, Dec. 1983 (to appear).

[6] D. Davies and J. F. Wakerly, "Synchronization and Matching in Redundant Systems," *IEEE Trans. Comput.,* Vol. C-27, No. 6, pp. 531-539, June 1978.

[7] L. Essen, "The Measurement of Frequency and Time Interval," *Her Majesty's Stationery Office,* London, 1973.

[8] A. L. Hopkins, *et al.,* "FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proc. IEEE,* Vol. 66, No. 10, pp. 1221-1239, October 1978.

[9] J. Goldberg, *et al.,* "Development and Analysis of the Software Implemented Fault-Tolerance (SIFT) Computer," *NASA CR-172140,* June 1983.

[10] J. H. Wensley, *et al.,* "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proc. IEEE,* Vol. 66, No. 10, pp. 1240 - 1255, October 1978.

[11] D. P. Sieworek, "Reliability Modeling of Compensating Module Failures in Majority Voted Redundancy," *IEEE Trans. Comput.,* Vol. C-24, No. 5, pp. 525-533, May 1975.

[12] T. B. Smith, "Fault-Tolerant Clocking System," *Digest of FTCS-11,* pp. 262 - 264, 1981.

[13] S. L. Hakimi and K. Nakajima, "Adaptive Diagnosis: A New Theory of t-Fault-Diagnosable Systems," *Twentieth Allerton Conf. Comm. Control, Comput.,* pp. 231-240, October 1982.

[14] A. W. Holt and J. M. Myers, "An Approach to the Analysis of Clock Networks," R-1280, *C.S.Draper Laboratory Contract Report (Preliminary)*, June 1978.

[15] N. A. Lynch, M. J. Fischer, and R. J. Fowler, "A Simple and Efficient Byzantine Generals Algorithm," *Proc. Reliability in Dist. Soft. and Database Syst.*, pp. 46 - 52, 1982.

[16] D. Palumbo and R. W. Butler, "SIFT -- A Preliminary Evaluation," *Proc. Fifth Digital Avionics Symp.*, Seattle, WA, August 1983.

[17] D. P. Siewerek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, MA, 1982.

[18] J. Losq, "A Highly Efficient Redundancy Scheme; Self-Purging Redundancy," *IEEE Trans. Comput.*, Vol. C-25, No. 6, pp. 569-578, June 1976.

[19] F. P. Mathur and A. Avizienis, "Reliability Analysis and Architecture of a Hybrid-Redundant Digital System: Generalized Triple Modular Redundancy with Self-Repair," *Spring JCC, AFIPS Conf. Proc.*, Vol. 36, pp. 373-383, 1970.

Drift Rate	Maximum Cluster Size
1×10^{-6}	43
5×10^{-6}	40
1×10^{-5}	40
2.2241×10^{-5}**	40
5×10^{-5}	37
1×10^{-4}	34
5×10^{-4}	22
1×10^{-3}	16

** Applies to SIFT.

Table 1. Maximum Cluster Size

Data Sample Period	m=1	m=2	m=3
100 ms	11.8 %	25.1 %	>380 %
50 ms	23.7 %	50.2 %	>760%
33 ms	35.9 %	76.2 %	>1140%
25 ms	47.4 %	>100 %	>1520%

m = number of faulty processors accommodated

Table 2. Interactive Consistency Overhead in SIFT.

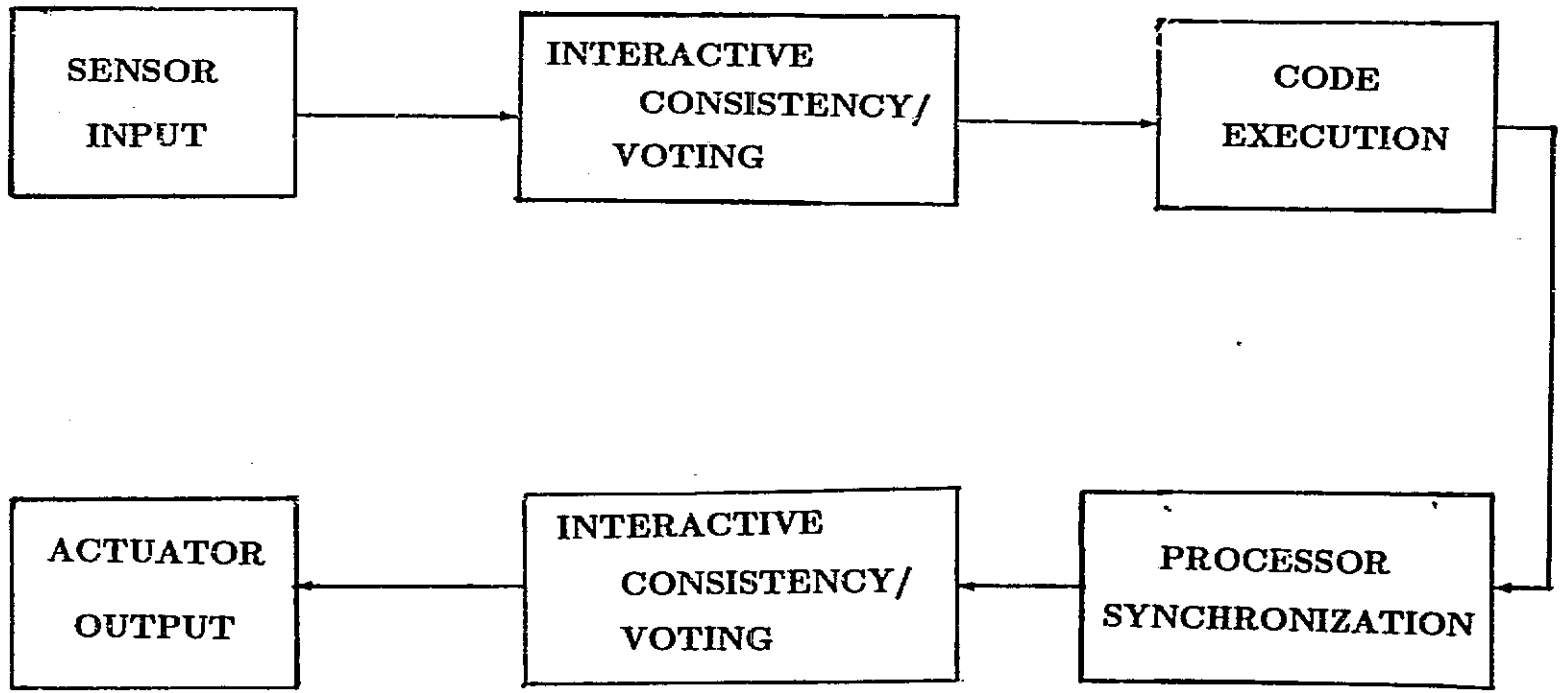


Figure 1. Functional block diagram of a real-time system

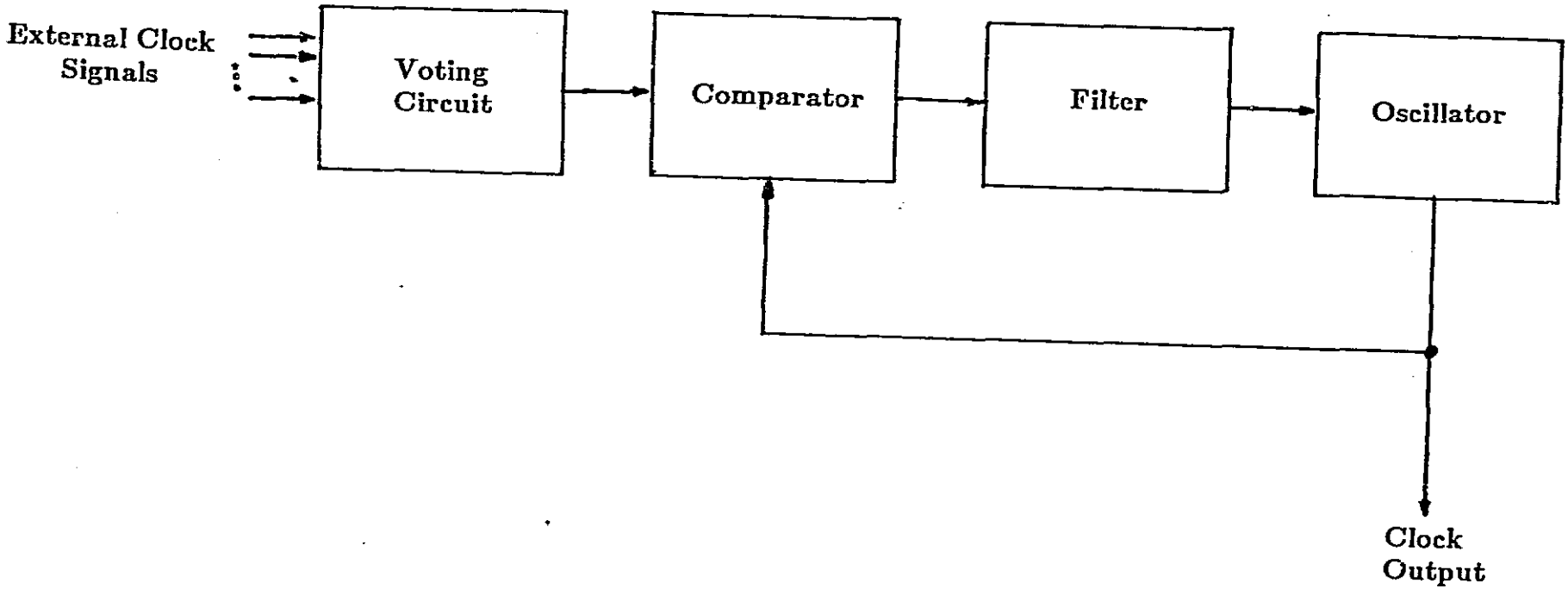


Figure 2. Component of a fault-tolerant phase-locked clock.

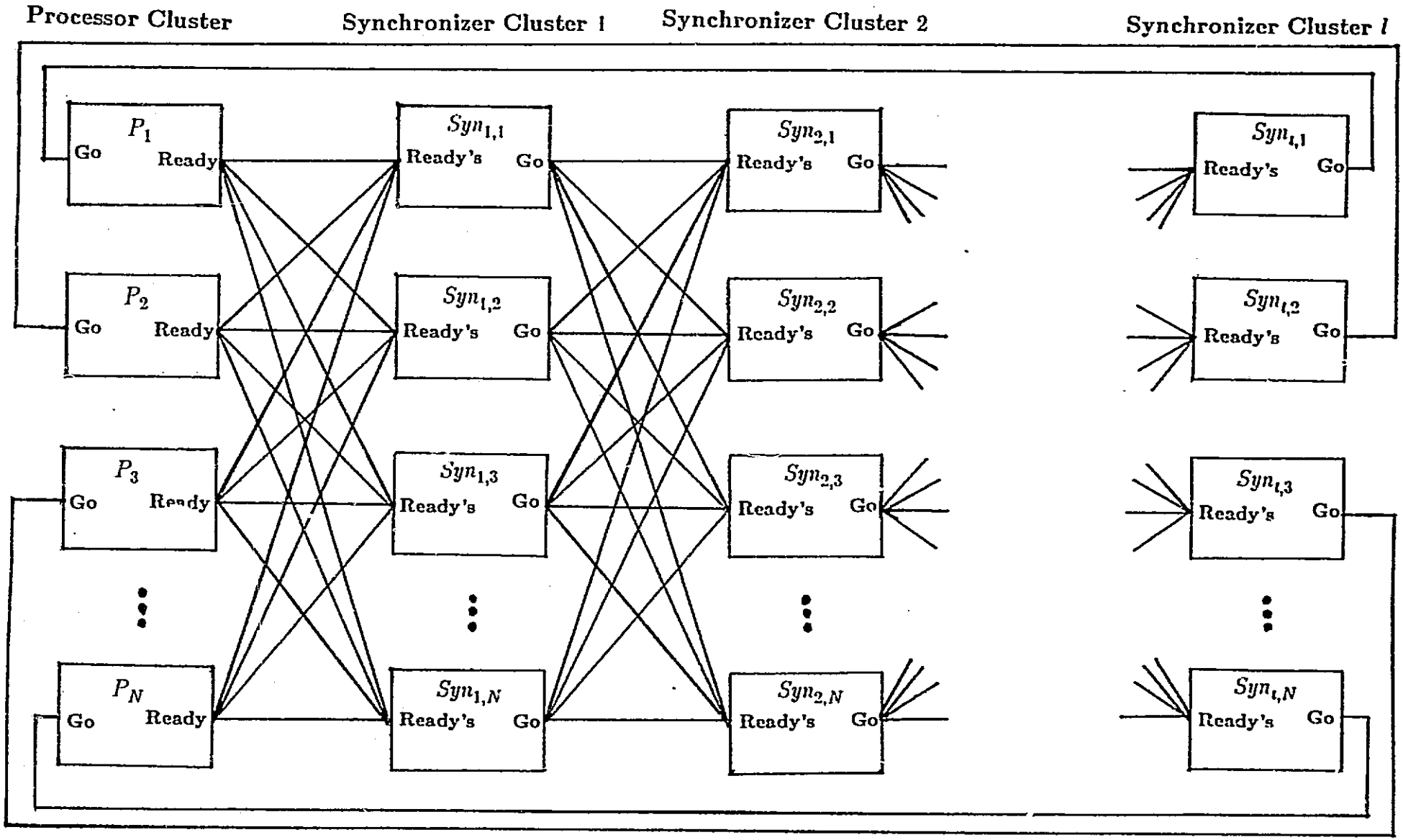


Figure 3. Davies & Wakerly's multistage synchronizer (from [6]).

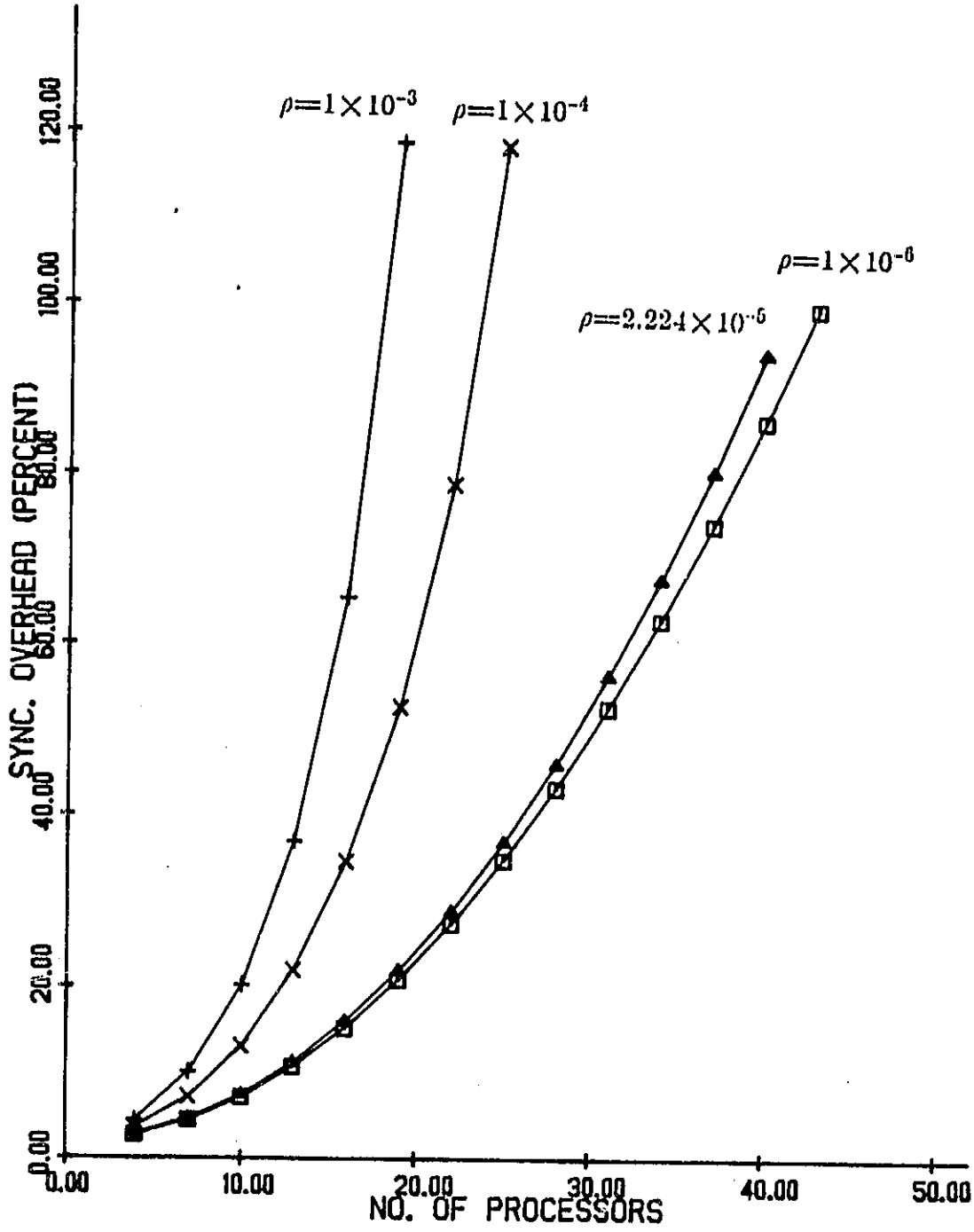


Figure 4. Overhead of Interactive Convergence Algorithm

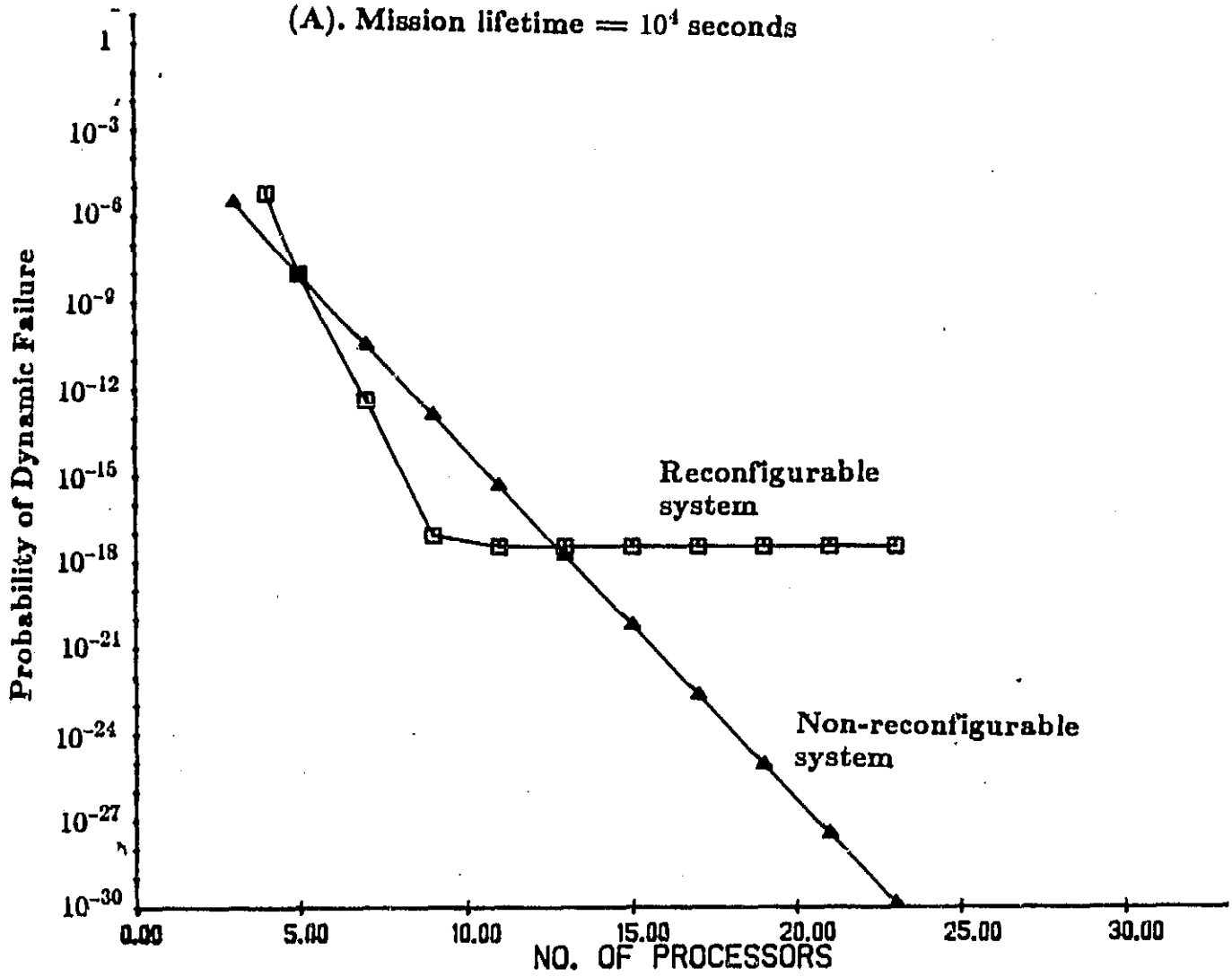
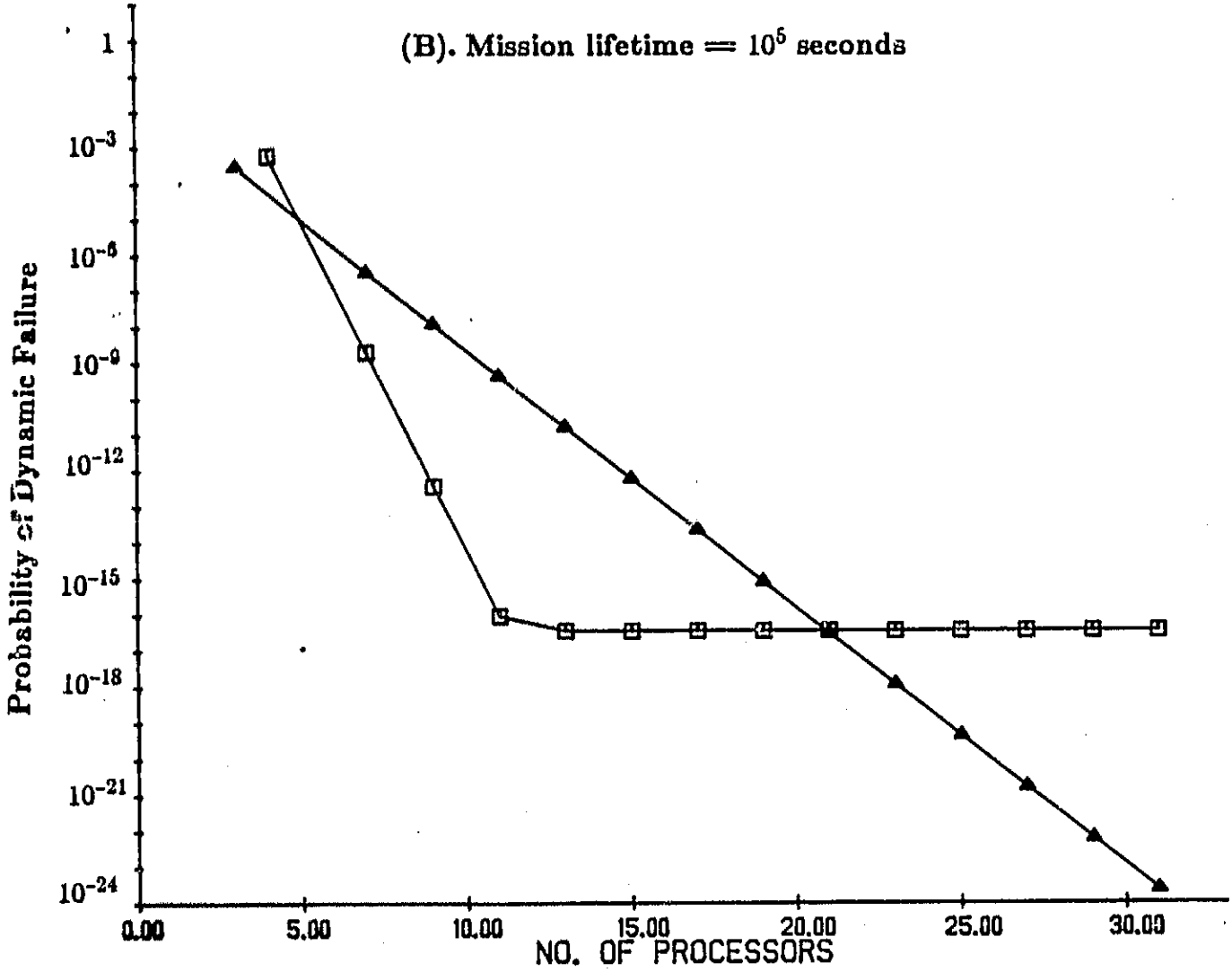
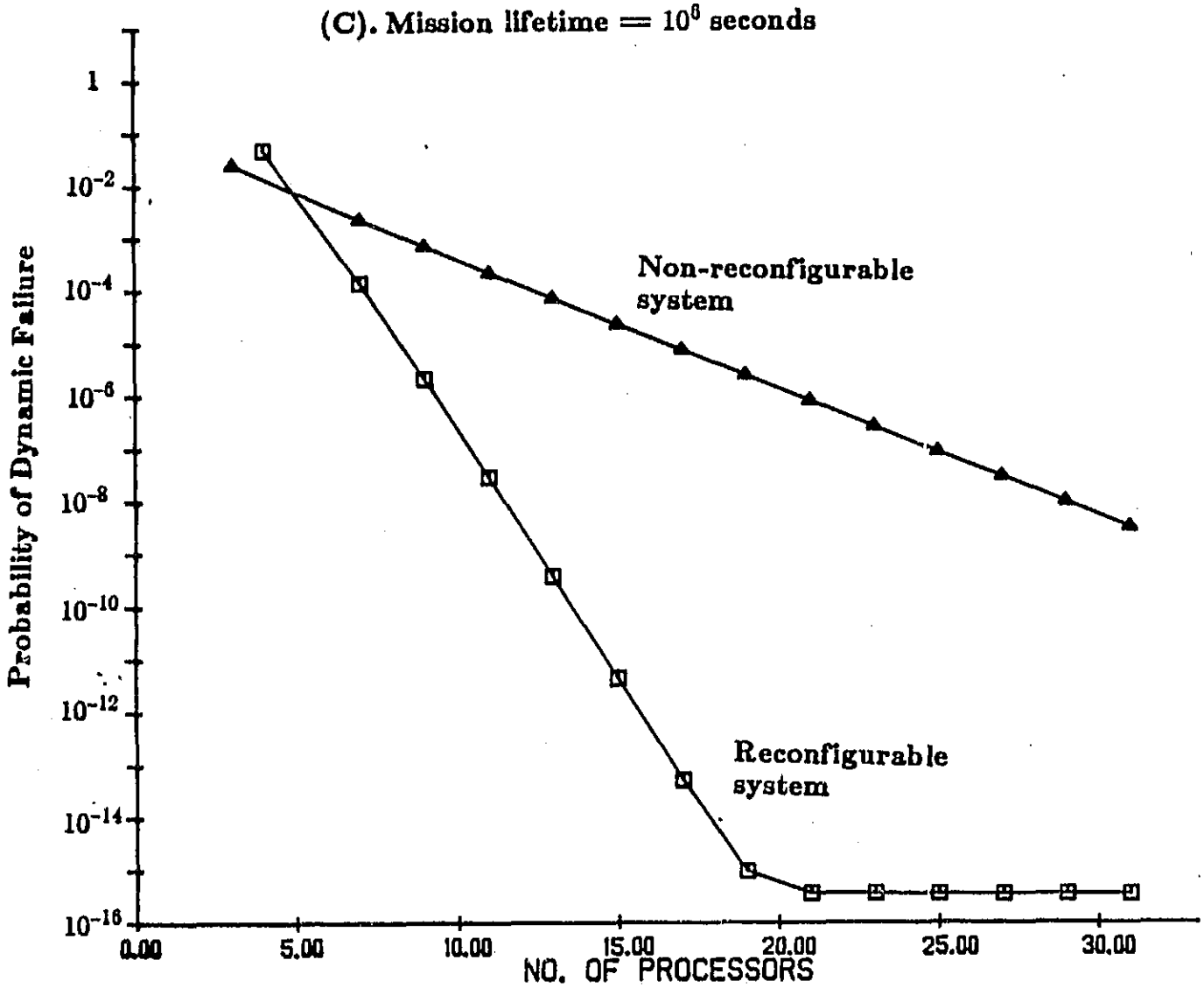


Figure 5. Comparison of reconfigurable and non-reconfigurable systems with maximum reconfigurable cluster size = 7, hard deadline = 100 milli-seconds, and task execution time = 3.2 milli-seconds, but with different mission lifetimes.


(B). Mission lifetime = 10^5 seconds
Probability of Dynamic Failure
1
10^{-3}
10^{-6}
10^{-9}
10^{-12}
10^{-15}
10^{-18}
10^{-21}
10^{-24}
0.00
5.00
10.00
15.00
20.00
25.00
30.00
NO. OF PROCESSORS


(C). Mission lifetime = 10^6 seconds
Probability of Dynamic Failure
1
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
Non-reconfigurable system
Reconfigurable system
0.00
5.00
10.00
15.00
20.00
25.00
30.00
NO. OF PROCESSORS