Clock Synchronization of a Large Multiprocessor System in the Presence of Malicious Faults

KANG G. SHIN, SENIOR MEMBER, IEEE, AND P. RAMANATHAN, STUDENT MEMBER, IEEE

Abstract—**Clock synchronization in the presence of malicious faults is one of the main problems associated with the design of a multiprocessor system. Although over the past few years many different algorithms have been proposed for overcoming this problem, they are not suitable for a large real-time multiprocessor system due to their excessive time overhead, asymmetric structure, and/or large number of interconnections.**

To remedy this problem, we propose a new method in this paper that i) requires little time overhead by using phase-locked clock synchronization, ii) needs a clock network very similar to the processor network, and iii) uses only 20-30 percent of the total number of interconnections required by a fully connected network for almost no loss in the synchronizing capabilities. Both ii) and iii) are made possible by grouping the various clocks in the system into many different clusters and then treating the clusters themselves as single clock units as far as the network is concerned. The method is significant in that regardless of their size multiprocessor systems can be built at an inexpensive cost without sacrificing both the synchronization and fault tolerance capabilities.

To show the feasibility of our method, an example hardware implementation is presented. This implementation turns out to be much simpler than the other existing methods and also retains the symmetry and synchronizing capabilities of the network.

Index Terms—**Byzantine Generals algorithm, clock synchronization, fault-tolerant real-time multiprocessors, malicious faults, phase-locked clocks, tick sequence.**

I. Introduction

CONTINUING advances in VLSI technology have made it attractive to build large multiprocessor systems by interconnecting hundreds or even thousands of inexpensive off-the-shelf microprocessors (with their own clocks) and memory modules. One of several advantages to be gained from such a large multiprocesor system is the high degree of multiprocessing: the multiprocessor system could execute many jobs in parallel. Each of these jobs is usually decomposed into a set of cooperating tasks that communicate closely with one another during the course of execution. These cooperating tasks are then assigned to a *group* of processors for execution which are often required to be tightly synchronized.

Manuscript received April 25, 1985; revised February 6, 1986 and May 20, 1986. This work was supported in part by NASA under Grants NAG-1-296 and NAG-1-492. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of NASA.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8611254.

This natural grouping of processors can be used to our advantage while trying to synchronize all the processors in the system, especially when some of the processors are *maliciously faulty*. A processor is said to be maliciously faulty if it *lies* by providing different information on the same object to different receivers. For example, maliciously faulty processors could try to prevent the nonfaulty processors in the multiprocessor system from synchronizing themselves by sending different time information about themselves to different nonfaulty processors during the course of synchronization. However, none of the existing clock synchronization algorithms [1]-[4] attempt to make use of the natural grouping mentioned above. Software synchronization algorithms such as CNV, COM, and CSM in [1] assume that there is a fully connected network of at least $3m + 1$ processors to tolerate up to m malicious failures. Then, each of these processors is asked to exchange their clock values periodically with other processors in order to synchronize themselves with the others. Since these synchronization algorithms treat all the processors in exactly the same manner, every nonfaulty processor is loosely synchronized with every other nonfaulty processor in the system. Consequently, it is not possible to run tasks that need a tight synchronization between the processors on which they run. In addition to this disadvantage, software synchronization algorithms also impose a high time overhead on the system performance, thereby making them unsuitable for real-time applications [5].

Hardware synchronization algorithms, on the other hand, impose no time overhead on the system performance. They, however, have peculiar problems of their own. For example, the algorithm proposed by Davies and Wakerly [6] requires a total of $2m^2 + 3m + 1$ devices (processors plus synchronizers) and $8m^3 + 16m^2 + 10m + 2$ I/O ports to be able to tolerate up to m malicious failures. The phase-locked algorithm [2], [3], [7] and Kessels' algorithm [4] neither require so many devices nor make use of the natural grouping of processors that arise due to the job partitioning. Instead, they need a fully connected network of $3m + 1$ clocks, which are then broadcast to every other clock in the network. This requires two different networks for clock synchronization: a fully connected network of $3m + 1$ clocks and a network of distributing these clock signals to the rest of the clocks in the system. This results in an asymmetric network of clocks, with some of them having a fan-out of N while the others have none where N is the total number of processors in the system. Moreover, the system interconnection becomes more complicated because such solutions need a clock network that would be completely *different* from the processor network in the system. It would be obviously preferable to have a clock network that is identical or similar to the processor network, since we could then easily implement the clock network by having some additional lines dedicated for transferring the clock signals on all the data paths.

For the reasons mentioned above, we need to develop a hardware synchronization algorithm that can synchronize the processors in the system at two different levels: one at the group level where all the processors operating on the same job or cooperating tasks are tightly synchronized with respect to each other and the other at the *system level* where any two processors belonging to two different groups (and thus operating on unrelated tasks) are loosely synchronized with respect to each other. The size of each group would be usually dependent on i) the number of cooperating tasks that are running simultaneously in that group and ii) the degree of redundancy required for reliability, i.e., each of the cooperating tasks will be executed on more than one processor in a group and individual execution results will be voted on. In other words, a processor group is composed of redundant clusters, each of which will execute the same task. Since the number of cooperating tasks varies from one job to another, Condition i) is usually not used for the determination of group size, e.g., *Cm**. On the other hand, the size of each cluster will be dependent only on the fault tolerance specification of the system. Hence, we shall use only Condition ii) for determining the cluster size and, thus, the group size.

Obviously, the processors within a cluster need a tighter synchronization among themselves than those in the same group. This leads to a need of three (instead of two) types of synchronization: tighter intracluster synchronization, tight intragroup synchronization, and loose intergroup synchronization. However, we shall err on the safe side by assuming that all the processors within a group need to be tightly synchronized as those in a cluster. In accordance with this assumption, the term "cluster" will henceforth be used to mean the same as the term "group."

One can minimize the total number of interconnections that are necessary to synchronize all the clocks in the system in such a way that the resulting clock network will be similar to the processor network in the system and satisfy the specified fault tolerance for each cluster. An interconnection scheme for achieving this objective is discussed in Section II.

The proof that the proposed interconnection scheme satisfies the synchronization conditions is given in Section III. We shall show that the maximum skew between any two nonfaulty clocks in the system will be less than or equal to 3δ where δ is the maximum skew between any two nonfaulty clocks within a cluster as a result of using an existing phase-locked algorithm. The algorithm to optimally partition the network in such a way that the total number of interconnections is minimized is described in Section IV. This algorithm could easily result in a 70–80 percent reduction in the total number of interconnections as compared to a fully connected network, especially when the specified fault tolerance is much less than $\lfloor N/3 \rfloor$. Even though this reduction is not as large as in [2], this scheme achieves the reduction while retaining the symmetry of the network and, therefore, is more useful for synchronizing large, real-time multiprocessor systems. Moreover, it is shown that the percentage difference in the number of interconnections between our method and that in [2] becomes very small as m increases. The hardware implementation of the above algorithm is described in Section V. The paper concludes with a summary of the results achieved by this algorithm in Section IV.

As can be seen easily, our method is ideally suited for synchronizing large multiprocessor systems in charge of time-critical applications, e.g., control of aircraft, nuclear reactors, industrial processes, and life-support systems that were addressed in [8].

II. The Proposed Architecture

Given a network of N processors, each of which has a clock of its own, the problem is to synchronize all the nonfaulty clocks in the network to a specified fault tolerance f_{spec} using as few interconnections as possible in such a way that the symmetry of the network is retained. For clarity of presentation, we begin with definitions of a few necessary terms below.

Definition 1: The time that is directly observable in some particular clock is called its *clock time*. This should be contrasted to the term *real time*, which is measured in an assumed Newtonian time frame that is not directly observable.

Definition 2: Let c be a mapping from clock time to real time, where $c(T) = t$ means that at clock time T, the real time is t. Then, two clocks c and c' are said to be *δ-synchronized* at a clock time T if and only if $|c(T) - c'(T)| \leq \delta$. It is customary to drop the T from the notation and write the condition as $|c - c'| \leq \delta$.

Definition 3: A set of clocks is said to be *well-synchronized* if and only if any two nonfaulty clocks of this set are synchronized to within a specified limit δ of each other.

Definition 4: A well-synchronized network has a *global clock cycle*. Global clock cycle i is the interval between the ith tick of the fastest nonfaulty clock (i.e., the nonfaulty clock that has its ith tick before that of all the other nonfaulty clocks) and the $i + 1$th tick of the fastest nonfaulty clock.

By "synchronize all the nonfaulty clocks in the network to a specified fault tolerance" we mean that in spite of having up to a specified number of faults in the network, the nonfaulty clocks in the system should remain well-synchronized. The phase-locked algorithm [3] achieves this objective, but at the cost of the network's symmetry and interconnection simplicity. To alleviate this problem, we shall develop a different interconnection strategy in Section II-A and a modified phase-locked algorithm in Section II-B.

A. The Interconnection Strategy

Since the processors within the same cluster operate on identical or related tasks, there is a need for tighter synchronization between the processors within the same cluster than between the processors belonging to different clusters. To meet this need, we synchronize the processors in the system by applying the phase-locked algorithm at two different levels. Each clock synchronizes itself not only with respect to all the clocks in its own cluster but also with respect to one clock from each of the other clusters by using the phase-locked algorithm. As a result of this mutual coupling between the clusters, these clusters remain synchronized with respect to one another and so the network as a whole remains well-synchronized.

Let M be the total number of clusters in this network and p_i be the total number of clocks in the ith cluster. Number the clusters in this network from 1 to M, and also number all the clocks in each cluster i from 1 to p_i. Let c_{ij} represent the jth clock of the ith cluster and let $q_{ik} \equiv [(i - 1) \bmod p_k] + 1$. Then each clock in the ith cluster receives as inputs not only all the clocks from its own cluster but also the q_{ik}th clock from each cluster $k \neq i$. This network architecture for $N = 8$, $M = 4$, and $p_i = 2$ for all i is shown in Table I, where a 1 in row i and column j indicates that the clock corresponding to column j is an input to the clock corresponding to row i.

There is no particular sanctity associated with this number q_{ik} as far as the algorithm is concerned. As long as each clock in the network receives a clock from every other cluster, the algorithm will work. However, the above formula ensures that the symmetry of the network is maintained because the formula results in similar fan-out for all the clocks in the network. We then claim that, along with the above architecture, if we use the phase-locked algorithm as described in the next subsection then the network as a whole will remain well-synchronized.

B. Modification of the Phase-Locked Algorithm

The phase-locked algorithm was first used (for a four-clock system) to ensure that all the processors of the fault-tolerant multiprocessor (FTMP) operate in lock-step [2]. The basic operation of this algorithm is as follows. Each individual clock is provided with an input receiver circuitry to receive all the clock pulses from the remaining clocks in its cluster and one clock from each of the other clusters. Each clock then uses these clock inputs to generate a reference signal. It then compares its own clock with the reference signal and computes an estimate of its own phase error. This phase error is then fed through a filter to a voltage controlled oscillator which then adjusts the frequency of its operation depending on the magnitude of the error. By adjusting the frequency of operation of each of these clocks with respect to one another, we can keep all of these clocks in lock-step with one another.

In [3], we have generalized the four-clock system [2], [7] into a system with an arbitrary number of clocks. However, there are two main differences between the present network and that in [3]:

- A given clock in the present network may receive inputs from clocks to which its own output is not connected. This was not possible in [3] because there every clock received inputs from every other clock in the network.
- As will be seen in Section IV, in the present network different clocks could receive different number of inputs. This again was not possible in [3], since every clock received exactly $N - 1$ inputs where N is the total number of clocks in the network.

These two differences cause little change in the phase-locked algorithm, since clock generates its reference signal only on the basis of the inputs it receives. It is independent of the reference signal of any other clock in the network. That is, if a clock receives I inputs (including itself), then it assumes there are I clocks in the network and functions accordingly. If the maximum number of faults to be tolerated is m, then we showed in [3] that i) $I > 3m$ and ii) the reference signal is generated as follows. Each clock first orders all the inputs it received in the order of arrival of the clock ticks. This ordered set is called the *tick sequence* of the corresponding clock. Let x be the position of its own clock in its tick sequence. Then the reference signal chosen by this clock is the $f_x(I)$th clock (excluding itself) in its tick sequence where $f_x(I)$ is any function satisfying the conditions described in [3]. This implies that if different clocks in our network receive different number of inputs then they will have a different function for generating the reference signal. This fortunately has no effect on the synchronizing capabilities of the network. We shall show in the following section that if δ is the maximum skew that can arise between any two nonfaulty clocks of a cluster as a result of applying the phase-locked algorithm then any two nonfaulty clocks in this network will be within 3δ of each other.

TABLE I
THE NETWORK INTERCONNECTIONS

TO	OUTPUTS FROM							
	c_{11}	c_{12}	c_{21}	c_{22}	c_{31}	c_{32}	c_{41}	c_{42}
c_{11}	1	1	1	0	1	0	1	0
c_{12}	1	1	1	0	1	0	1	0
c_{21}	0	1	1	1	0	1	0	1
c_{22}	0	1	1	1	0	1	0	1
c_{31}	1	0	1	0	1	1	1	0
c_{32}	1	0	1	0	1	1	1	0
c_{41}	0	1	0	1	0	1	1	1
c_{42}	0	1	0	1	0	1	1	1

III. The Design Proof

Consider a clock of the network formed by a set of clocks CK. There are two possibilities: either this clock is connected only to all the clocks of its own cluster or it is also connected to at least one cluster other than its own. This fact leads to the decomposition of CK into two subsets A and B such that $A \cap B = \emptyset$, A is the set of all clocks which are connected to at least one cluster other than its own cluster, and B is the set of all clocks connected only to its own cluster. Now let CL_i, $i \in L \equiv \{1, 2, \cdots, M\}$, be one of the M clusters forming the network. Due to the interconnection strategy we have adopted, for any cluster pair CL_s and CL_m of this network, there is one and only one clock in CL_s which serves as an input *to* all the clocks in CL_m. Denote this clock by the ordered pair (s, m). Note that there is one more clock link between CL_s and CL_m, but this clock is *from* CL_m *to* CL_s, i.e., it is the input to CL_s from CL_m, and so will be denoted by $(m, s) \in CL_m$. Also note that every such ordered pair of clusters uniquely represents a clock in set A. On the other hand, a clock in A

can have more than one such ordered pair representation but definitely has one such representation. Based on this observation, we can partition the clocks into four groups with respect to any given cluster CL_m as follows (see Fig. 1):

$$IN_m \equiv \{s : s \in L, \quad (s, m) \text{ is nonfaulty}\}$$
$$IF_m \equiv \{s : s \in L, \quad (s, m) \text{ is faulty}\}$$
$$ON_m \equiv \{s : s \in L, \quad (m, s) \text{ is nonfaulty}\}$$
$$OF_m \equiv \{s : s \in L, \quad (m, s) \text{ is faulty}\}.$$

If $i = k$, then c_{ij} and c_{kl} are two nonfaulty clocks of the same cluster, and hence are within δ of each other by definition. Thus, the more interesting case is when $i \neq k$.

Henceforth, we proceed to show that irrespective of the distribution of the faults in the network there exists a nonfaulty link from either CL_i to CL_k or vice versa with at most two hops.

Lemma: For any two clusters CL_i and CL_k satisfying $|OF_i| < |IN_k|$, there exists a nonfaulty path from cluster CL_i to cluster CL_k with at most two hops (see Fig. 2) where $|D|$ is the cardinality of the set D.

Proof: First, suppose that the clock (i, k) is nonfaulty. Then, irrespective of the way the other faults are distributed we have a direct path from CL_i to CL_k containing exactly one hop. Therefore, the more interesting case occurs when the clock (i, k) is faulty. In this case, we shall show via contradiction that there exists a nonfaulty path from CL_i to CL_k containing exactly two hops.

Assume that there exists no cluster $CL_q \in IN_k$ such that (i, q) is nonfaulty. This means that for any cluster $CL_s \in IN_k$ the clock (i, s) is faulty. Thus, there are at least $|IN_k|$ faulty clock outputs from CL_i, i.e., $|OF_i| \geq |IN_k|$, a contradiction. ■

Theorem: Let c_{ij} and c_{kl} be any two nonfaulty clocks of the network. Also let δ be the maximum skew that can arise between any two nonfaulty clocks in a cluster as a result of applying the phase-locked algorithm. Then, $|c_{ij} - c_{kl}| \leq 3\delta$ for all i, j, k, and l under the condition $p_{\max} \leq 2M - 2$ where $p_{\max} = \max(p_1, p_2, \cdots, p_M)$.

Proof: Let $|IF_k| = x$ and f be the specified fault tolerance of the network. Since there are a total of M clusters in this network, we know from our interconnection strategy that the total number of external inputs to any cluster is $M - 1$. That is,

$$|IF_k| + |IN_k| = M - 1$$
$$\text{or } |IN_k| = M - x - 1 \quad \text{for all } k \in L. \quad (3.1)$$

Consider the following two cases.

Case 1: $|OF_i| < \min(M - x - 1, f + 1)$.

Since $|OF_i| < M - x - 1 = |IN_k|$, by lemma, even at worst, there exists a cluster CL_q such that clocks (i, q) and (q, k) are both nonfaulty. From the triangle inequality we get

$$|c_{ij} - c_{kl}| \leq |c_{ij} - (i, q)| + |(i, q) - (q, k)| + |(q, k) - c_{kl}| \leq 3\delta.$$

Case 2: $M - x - 1 \leq |OF_i| \leq f$.

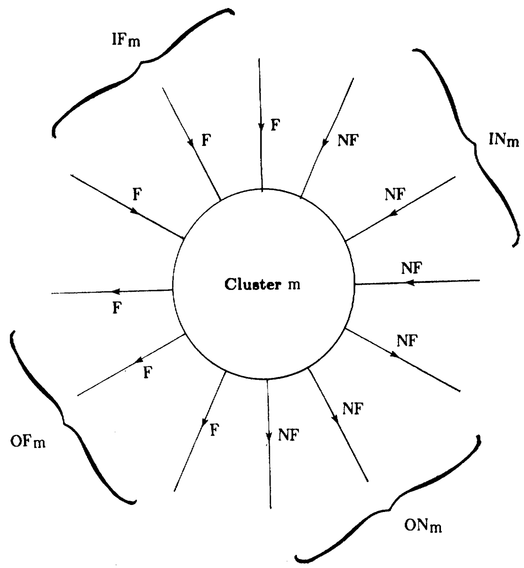


Fig. 1. Partition of network based on cluster CL_m.

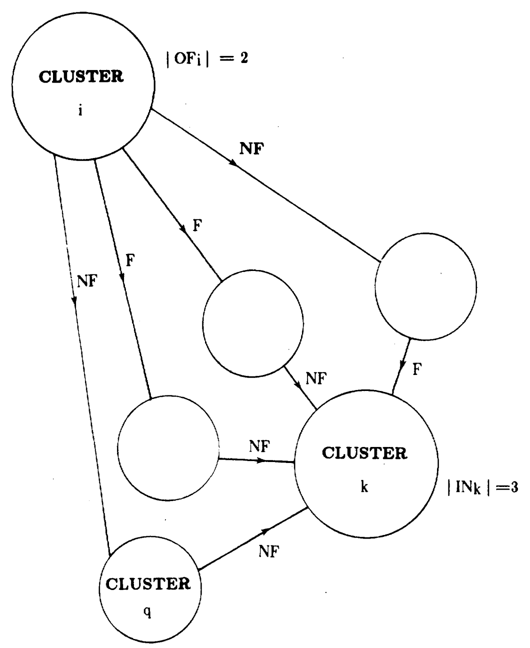


Fig. 2. The case of $|OF_i| < |IN_k|$.

It is now possible that there is no $CL_q \in IN_k$ such that (i, q) is nonfaulty, i.e., there is no nonfaulty link from CL_i to CL_k (for example, see Fig. 3). In such a case, we shall show that there is always a nonfaulty path from CL_k to CL_i.

Let $r \equiv \lceil M/p_{\min} \rceil$ where $\lceil x \rceil$ is the smallest integer not less than x and $p_{\min} = \min(p_1, p_2, \cdots, p_M)$. Then, according to our interconnection strategy, every clock in this network could go to at most r different clusters. Using this with the fact that $M - x - 1 \leq |OF_i|$, there are at least $\lceil (M - x - 1)/r \rceil$ faulty clocks in CL_i. Thus, there are at most $f - \lceil (M - x - 1)/r \rceil$ faulty clocks in the inputs to CL_i, i.e.,

$$|IF_i| \leq f - \left\lceil \frac{M - x - 1}{r} \right\rceil$$
$$\text{or } |IN_i| \geq M - 1 - f + \left\lceil \frac{M - x - 1}{r} \right\rceil . \quad (3.2)$$

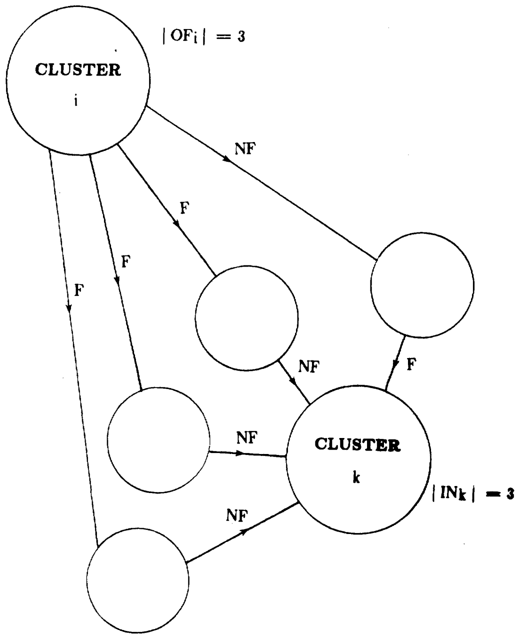


Fig. 3. The case of $|OF_i| \geq |IN_k|$.

Since $|IF_k| = x$, there are at most $f - x$ faulty clocks in CL_k, and thus

$$|OF_k| \leq r(f-x). \tag{3.3}$$

Now, if possible, let there be no nonfaulty path from CL_k to CL_i of length less than or equal to two hops. Then, by the proof of the lemma $|OF_k| \geq |IN_i|$, and therefore, from (3.2) and (3.3) we get

$$r(f-x) \geq (M-1) - f + \left\lceil \frac{(M-x-1)}{r} \right\rceil$$

$$\geq (M-1) - f + \frac{(M-x-1)}{r} \tag{3.4a}$$

or

$$r^2 f + rf \geq (r+1)(M-1) + (r^2-1)x. \tag{3.4b}$$

Since the maximum value that x can take is f, we get $f \geq M - 1$. Now from our interconnection strategy we know that the maximum number of inputs to any clock is $M + p_{\max} - 1$ where $p_{\max} = \max(p_1, p_2, \cdots, p_M)$. From the Byzantine Generals paradigm we get $M + p_{\max} - 1/3 > f$, leading to

$$M - 1 \leq f < \frac{M + p_{\max} - 1}{3} \quad \text{or} \quad p_{\max} > 2M - 2 \tag{3.5}$$

which is contradictory to our hypothesis. Therefore, $|OF_k| < |IN_i|$, and so by the lemma there exists at least one $q \in IN_i$ such that the clock (k, q) is nonfaulty. That is, for all i, j, k, and l

$$|c_{kl}| \leq |c_{kl} - (k, q)| + |(k, q) - (q, i)| + |(q, i) - c_{ij}| \leq 3\delta.$$

■

The condition $p_{\max} \leq 2M - 2$ in the above theorem implies that the connectivity of the network is sufficiently large to ensure a good synchronization. Ideally, to have a maximum fault tolerance, we would like to have a fully connected network, i.e., $p_{\max} = 1$ and $M = N > 1$, in which case the above condition is obviously satisfied. However, since we cannot afford to have so many interconnections in the network, we have to compromise on the maximum achievable fault tolerance of the network. We shall show in the next section that it is sufficient to have $p_{\max} \leq \sqrt{N}$ to ensure least number of interconnections under any fault tolerance specification when all the clusters are of the same size. The above condition is a slight generalization of this condition because $p_{\max} \leq \sqrt{N}$ implies that $p_{\max} \leq 2M - 2$.

IV. Minimization of the Number of Interconnections

Assume for the time being that $p_{\min} = p_{\max} = p$, i.e., all the clusters have the same size, and thus $Mp = N$ where $N = |CK|$. Our aim is to minimize the total number of interconnections $J \equiv Mp(p-1) + Mp(M-1) = N(M+p-2)$ subject to the required level of fault tolerance, which can be stated as $M + p - 2 \geq 3f_{\text{spec}}$ from the Byzantine Generals paradigm and our interconnection strategy. Substituting for p in $M + p - 2$ from $N = Mp$ and differentiating with respect to M shows that $M + p - 2$ increases with M. Therefore, both the total number of interconnections and the fault tolerance of the network increase with M. So minimizing J is the same as minimizing M. By solving for M in the fault tolerance condition and combining it with the above result, we can easily get a unique value for M that minimizes the total number of interconnections while ensuring that the fault tolerance requirement is met.

However, the assumption that all clusters are of the same size is not suitable for all values of N. For example, if N were a prime number, then we would not be able to find two factors M and p other than N and 1. This means that we can get only a fully connected network if we restrict ourselves to clusters of single size. On the other hand, any N can be decomposed into clusters of two different sizes,[1] say p_1 and p_2. This will result in a network which has fewer interconnections than a fully connected network. We shall therefore devise an algorithm to divide the clocks into clusters of two different sizes such that the total number of interconnections is minimized.

A. Optimization for Two Different Cluster Sizes

Let the total of N clocks be divided into M_1 clusters of p_1 clocks each and M_2 clusters of p_2 clocks each, where $p_1 \geq p_2$, i.e., $M_1 p_1 + M_2 p_2 = N$. The scheme of interconnection is the same as explained in Section II. First, number the clusters with p_1 clocks (from now on referred to as CP1) from 1 to M_1 and the clusters with p_2 clocks (CP2) from $M_1 + 1$ to $M_1 + M_2$. Also number the clocks in a given cluster from 1 to p_1 or 1 to p_2 correspondingly.

Let $q_{i1} = [(i-1) \bmod p_1] + 1$ and $q_{i2} = [(i-1) \bmod p_2] + 1$. Then each clock in cluster i receives the q_{i1}th clock from each CP1 and q_{i2}th clock from each CP2 in addition to all the clocks from its own cluster. Then, as in the earlier case, each clock uses the phase-locked algorithm to synchronize itself with the rest of the clocks at its input.

[1] In fact, there is no need to consider clusters of more than two different sizes. More on this will be discussed later.

The problem is now to determine M_1, p_1, M_2, and p_2 that minimize the total number of interconnections and meet the specified fault tolerance requirement. The solution to this problem is more difficult than in the earlier case, because we now have three independent variables M_1, p_1, p_2 in contrast to one in the other case.

The total number of interconnections in this network can be derived easily as follows:

- The total number of inputs to each clock in CP1: $M_1 + M_2 + p_1 - 1$.
- The total number of inputs to each clock in CP2: $M_1 + M_2 + p_2 - 1$.

Consequently, we derive

$$J = M_1 p_1 (M_1 + M_2 + p_1 - 1) + M_2 p_2 (M_1 + M_2 + p_2 - 1)$$
$$= N(M_1 + M_2 - 1) + M_1 p_1^2 + M_2 p_2^2. \tag{4.1}$$

The decomposition problem can be stated formally as follows.

Problem D: Determine nonnegative integers M_1, p_1, M_2, p_2 which minimize

$$J = N(M_1 + M_2 - 1) + M_1 p_1^2 + M_2 p_2^2$$

subject to

$$M_1 p_1 + M_2 p_2 = N \tag{4.2}$$

$$M_1 + M_2 + p_2 - 2 \geq 3 f_{spec} \tag{4.3}$$

$$p_1 - p_2 \leq 0 \tag{4.4}$$

$$p_1 \leq 2(M_1 + M_2 - 1). \tag{4.5}$$

Since there are only finitely many integers between 0 and N, there are only finitely many possible solutions for M_1, M_2, p_1, p_2. Thus, there definitely exists an integer solution to the above problem. For small N we can actually determine the solution by enumerating all the possible solutions and choosing the one that gives the minimum value for J. But the complexity of this solution process is $O(N^3)$, making it unacceptable for a large N. In such situations we can take recourse to the standard nonlinear integer programming methods like the generalized reduced gradient method with branch-and-bound principle [9].

However, the solution to this problem requires a large number of calculations. A nonlinear integer programming problem has to be solved repeatedly until an all integer solution is obtained. If we had to use three or more different sizes instead of two, then the calculation would have become much more complicated because we usually have to solve the problem many more times. However, we do not have to consider clusters of more than two different sizes. This can be justified as follows: As p_1 and p_2 decrease in magnitude, the network tends to change towards a fully connected network, i.e., the fault tolerance of the network increases and so does the total number of interconnections. These are the two opposing factors which determine the values of p_1 and p_2. Had there been no fault tolerance condition the minimum number of interconnections would have occurred at $p_1 = p_2 = \sqrt{N}$. From symmetry considerations we know that even in this case, the minimum number of interconnections will occur when $p_1 = p_2$. But since for any general N we may not be able to find such a p_1 and p_2, minimum number of interconnections occur when $p_1 - p_2 = 1$. Since it is possible to decompose any N into this form, at worst, we need clusters of two different sizes p_1 and $p_1 - 1$.

TABLE II
VARIATIONS WITH RESPECT TO SIZE OF THE NETWORK

N	f_{spec}	M_1	p_1	M_2	p_2	J	% Reduction
20	3	2	3	7	2	206	45.79
30		6	5	0	0	300	65.52
40		2	6	4	7	468	70.00
50		1	8	6	7	658	73.14
62		2	7	6	8	916	75.78
64		1	8	7	8	960	76.19
100		10	10	0	0	1900	80.19
-							
20	5	4	2	12	1	328	13.68
30		1	2	14	2	480	44.83
40		5	2	10	3	670	57.05
50		6	3	8	4	832	66.04
62		2	6	10	5	1004	73.45
64		4	6	8	5	1048	74.01
100		10	10	0	0	1900	80.19
-							
20	7	-	-	-	-	-	-
30		14	1	8	2	676	22.30
40		4	1	18	2	916	41.28
50		8	3	13	2	1124	54.12
62		2	4	18	3	1372	63.72
64		4	4	16	3	1424	64.68
100		8	5	10	6	2260	77.17

In any case, there exists a unique network architecture which minimizes the total number of interconnections while retaining the symmetry of the network. We shall next illustrate this with a few numerical examples.

B. Numerical Examples

The algorithm described above to determine the values of M_1, p_1, M_2, and p_2 has been implemented in Pascal on a DEC VAX®-11/780 and the following results were obtained.

First, we want to determine the variation of the total number of interconnections with respect to the size of the network when the fault tolerance requirement is kept constant. The values obtained for the fault tolerance $f_{spec} = 3$, 5, and 7 are given in Table II and the plot of the variation is given in Fig. 4. It also gives the percentage reduction in the total number of interconnections required by the interconnection scheme proposed in this paper as compared to a fully connected network.

Fig. 4 indicates that for a given fault tolerance condition the total number of interconnections increases proportionately to the size of the network. This is because when the size of the network increases, the number of clocks in the network increases, and so obviously we need more interconnections to keep them synchronized. However, even though the total number of interconnections increases, the percentage reduction in the total number of interconnections also increases with the size of the network. This is because typically the fault tolerance condition will not increase proportionately to the size of the network we will encounter. Therefore, for the same fault tolerance condition we will be working with larger and larger networks. In such a case, reduction in the total number of interconnections will be one of our main criteria for which our algorithm works well.

Another fact that is obvious from Table II is that reductions

®VAX is a registered trademark of Digital Equipment Corporation.

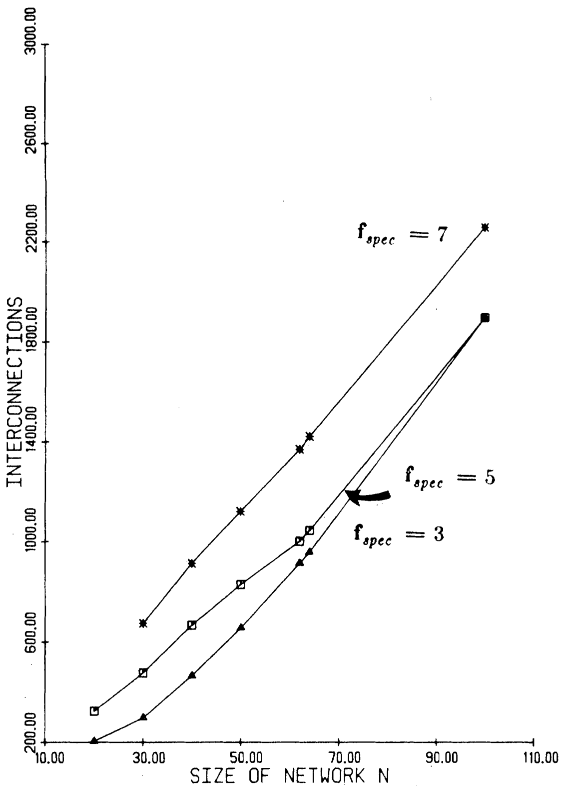


Fig. 4. Total number of interconnections versus size of network.

up to 80 percent can be easily achieved for relatively less stringent fault tolerance condition. However, in such situations, the interconnection scheme used in [2] requires much less number of interconnections as compared to the interconnection scheme proposed in this paper (see Table III). But as the fault tolerance specification becomes more stringent the percentage difference between the number of interconnections in both schemes drops rapidly to a very low value and then remains almost constant through the entire range. This fact is clearly depicted by the plots in Fig. 5. In other words, the scheme used in [2] is only marginally better than the scheme in this paper under almost all fault tolerance specifications. Thus, the scheme proposed in this paper maintains the synchronizing capabilities of the network, and achieves the symmetry of the network as well as a considerable reduction in the total number of interconnections as compared to a fully connected network.

TABLE III
VARIATIONS WITH RESPECT TO SPECIFIED FAULT TOLERANCE

N	f_{spec}	M_1	p_1	M_2	p_2	J	J_{FTMP}	% Increase
20	2	5	4	0	0	160	120	33.3
	3	2	3	7	2	206	180	14.44
	4	6	1	7	2	274	240	14.17
	5	4	2	12	1	328	300	9.33
	6	20	1	0	0	380	360	5.55
62	3	2	7	6	8	916	558	64.16
	5	2	6	10	5	1004	930	7.9
	7	2	4	18	3	1372	1302	5.38
	8	10	2	14	3	1592	1488	6.99
	9	8	3	19	2	1760	1674	5.14
	12	12	1	25	2	2344	2232	5.01
	18	7	2	48	1	3424	3348	2.27
	20	20	1	0	0	3782	3720	1.67
64	3	8	8	0	0	960	576	66.67
	5	4	6	8	5	1048	960	9.17
	7	4	4	16	3	1424	1344	5.95
	9	10	3	17	2	1822	1728	5.44
	12	10	1	27	2	2422	2304	5.12
	18	9	2	46	1	3538	3456	2.37
	21	64	1	0	0	4032	4032	0.
100	3	10	10	0	0	1900	900	111.11
	6	10	10	0	0	1900	1800	5.55
	10	16	3	13	4	3152	3000	5.07
	15	10	3	35	2	4630	4500	2.88
	20	22	1	39	2	6178	6000	2.97
	25	24	2	52	1	7648	7500	1.97
	30	9	2	82	1	9118	9000	1.31
	33	100	1	0	0	9900	9900	0.

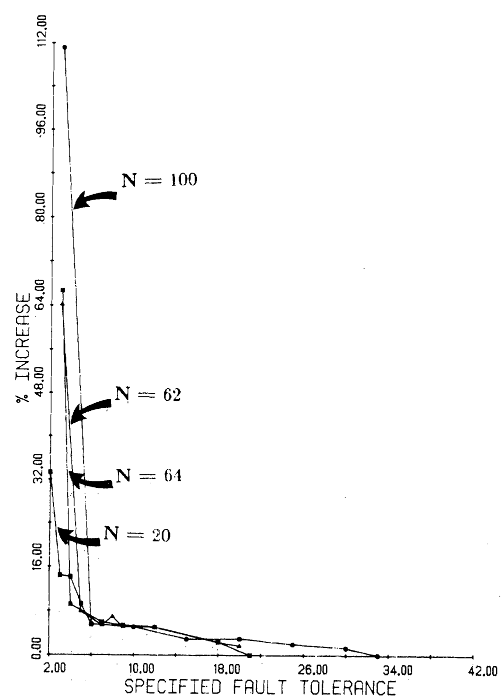


Fig. 5. Percentage increase versus specified fault tolerance.

V. Hardware Implementation

The phase-locked algorithm requires that every clock in the network be able to perform the following two operations:

1) take in the values of every clock at its input and generate an appropriate reference clock signal, and

2) use this reference signal to adjust its frequency, if necessary.

For this purpose, every clock in the network is provided with a synchronization circuitry whose block diagram for a clock s is given in Fig. 6. Blocks A, B, and C are used to generate the reference signal, whereas the Block D is used for adjusting the frequency whenever necessary. The design of Block D is very simple, since it just consists of a phase detector to detect the phase error between reference signal and the clock s. This error signal is then fed through a low-pass filter to a voltage controlled oscillator, which adjusts the frequency depending on the magnitude of the error. On the other hand, the design of the other three blocks is not so simple. The rest of this section will concentrate on discussing their design.

Block A—The Clock Input Circuitry: Fig. 7 shows the input circuitry of clock s. It receives the clock values from the other input clocks through a set of D-type FF's. In addition to these FF's, this block consists of n k-bit registers, a k-bit high-frequency counter, and a high-frequency clock C_{hf} where n is the total number of inputs to this clock including itself. The clock C_{hf} is directly connected to the high-frequency

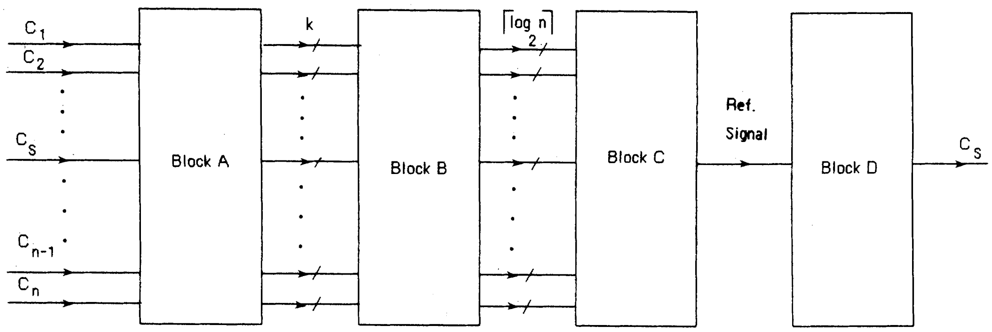


Fig. 6. Block diagram of clock synchronization circuitry.

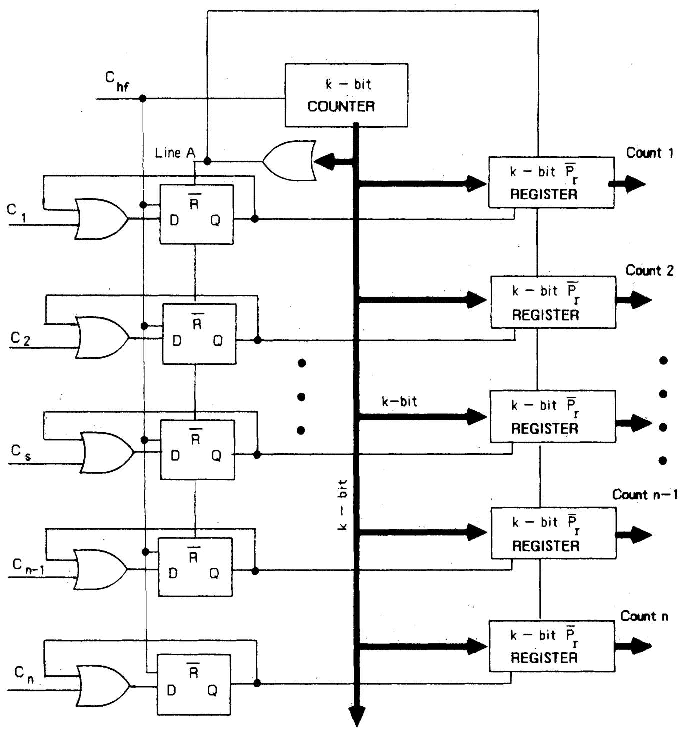


Fig. 7. Block A—clock input circuitry.

counter, making the counter count up. While all the registers and the flip-flops are reset at the beginning of every global clock cycle (*gcc*), the counters are preset to all ones. At a trailing edge of the clock pulse C_{hf}, an incoming clock signal will set the corresponding *FF*. Once the *FF*'s get set, they will remain set until they are explicitly reset before the commencement of the next *gcc*. This prevents a faulty clock from providing multiple transitions during the same *gcc*.

When the *FF* associated with a particular clock gets set, it also clocks in the current counter value into the k-bit register associated with that clock. As a result of this, the arrival of each tick is now marked by a k-bit number, which represents the relative ordering of the clocks as per their arrival. The clock with the least number in the register is the fastest clock in this *gcc*, whereas the clock with the largest number is the slowest. The intermediate clocks are also ordered in the increasing order of the numbers. If two clocks arrive within the same clock period of C_{hf}, then both the clocks are considered to have arrived at the same time and hence will have the same number in their registers.

The next *gcc* begins when the counter value goes back to zero. The modulo of the counter is so chosen that the counter goes through exactly one cycle of all the states during one *gcc*. The maximum frequency of the high-frequency clock, and hence the maximum resolution between two input clocks is determined by the maximum propagation delay in the various components in this block. The frequency should be low enough to ensure that the corresponding counter value is clocked into the appropriate register before the arrival of the next clock pulse.

Knowing both the time period of C_{hf}, T, and the desired global clock time period, T_{gcc}, we can easily determine the modulo of the counter as T_{gcc}/T, which in turn determines the value of k.

Thus, given T_{gcc} and the component delays, we can completely determine the design parameters of this block. The n k-bit words serve as the outputs of this block. They are directly fed into Block B, where they are sorted in the order of arrival of the ticks.

Block B—Tick Sequence Generator: (see Fig. 8) This block takes in as input all the register values from Block A and arranges them in the ascending order, i.e., the tick sequence.

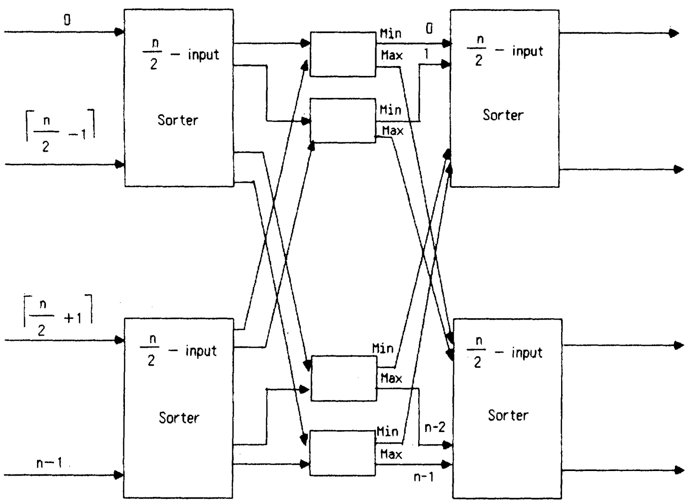
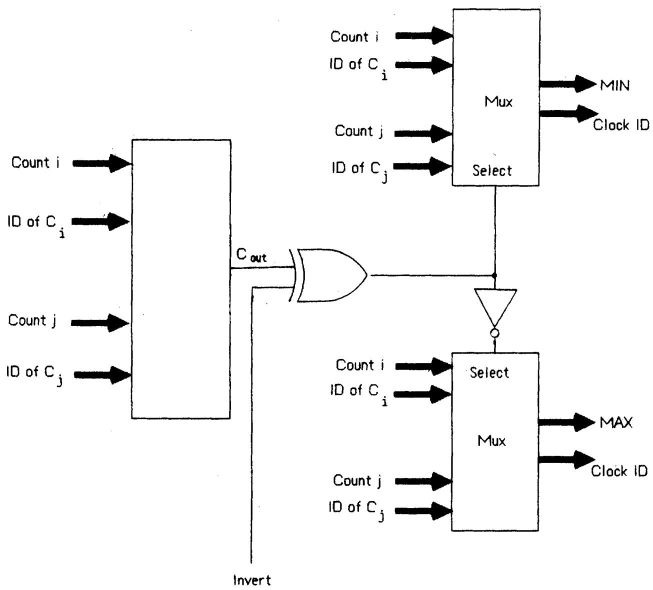


Fig. 8. Block B—tick sequence generator.



$$C_{out} = G_k + P_k G_{k-1} + P_k P_{k-1} G_{k-2} + \cdots + P_k P_{k-1} \cdots P_1 G_0$$

$$G_i = \overline{A_j + B_i}$$

$$P_i = \overline{A}_j B_i$$

Fig. 9. Comparison and exchange module.

This sorting of the numbers is done by a network of comparison and exchange modules. Each comparison and exchange module acts as a 2-input sorter. It has two inputs and two outputs. Each input has two parts in it; a register value and the ID of the clock that had this register value. Each module compares (via subtraction) the two register values at its input and then outputs the lesser of the two along with its clock ID on the min lines and the greater of the two and the corresponding clock ID on the max lines. The internal details of a 2-input sorter is shown in Fig. 9. By performing a series of such comparison and exchanges, we can arrange all the n inputs to this block in an ascending order [10]. Using a 4-input sorter and a 2-input sorter as the basic building blocks, we can iteratively build a 2^k-input sorter for any k, as shown in Fig. 8.

The complexity and the delay involved in this block can be easily analyzed. However, before going into that, consider briefly how Block C selects the appropriate signal based on the tick sequence it receives from Block B.

Block C—Reference Signal Selector: The internal details of this block is shown in Fig. 10. This block receives the tick sequence generated by Block B and then selects the appropriate reference signal, i.e., implementation of the $f_x(N)$ function in Section II-B.

The selection of the reference signal is done on the basis of the position x of its own clock in the tick sequence as shown below:

begin
if $x \geq N - m$ then choose the $m + 1$th clock in the tick sequence
else if $x \leq 2m$ then choose the $2m + 1$th clock in the tick sequence
else choose the $2m$th clock in the tick sequence
end.

To determine which position its clock is actually in, the clock ID inputs from the Block B are decoded to check for its own ID. Based on the result of this test, one of the three multiplexers gets enabled and the appropriate reference clock is selected. This reference signal is then input to Block D for the frequency adjustments.

A. Delay Analysis of the Circuit

The parameter of our major design concern is the delay associated with this entire synchronization circuitry. For the ease of explanation, we shall calculate this delay for a particular example of $n = 8$. The extension of this calculation for a larger n is trivial. We also assume some typical delays associated with each component in the circuit based on the manufacturer's specification. For example, the delay associated with a comparison and exchange module is taken to be 50 ns, that of multiplexers is taken to be 15 ns and that of other gates, flip-flops, etc., are taken to be 7 ns. Obviously, the main delay occurs in Block B. Block B in our example has eight stages of comparison and exchange modules. Therefore, a maximum delay of less than 400 ns will occur in Block B. The delay in Block A is not of concern to us because we are interested in the delay in generating the reference signal once the clock pulses have actually arrived. Block C consists of a few gates and a multiplexer. So, at worst, it takes about 100 ns in Block C and so it takes less than 500 ns to generate the reference signal, once at least n–m clock pulses have arrived in Block A. This means that if we are operating at 1 MHz global clock frequency, we will have our reference signal ready well within one global cycle. Hence, we will be able to update the reference signal once every global clock cycle. This means that the frequency adjustment in the current cycle is made on the basis of the position of the clocks in the previous cycle. This is perfectly reasonable since the clocks are unlikely to change their frequency of operation substantially within one clock period.

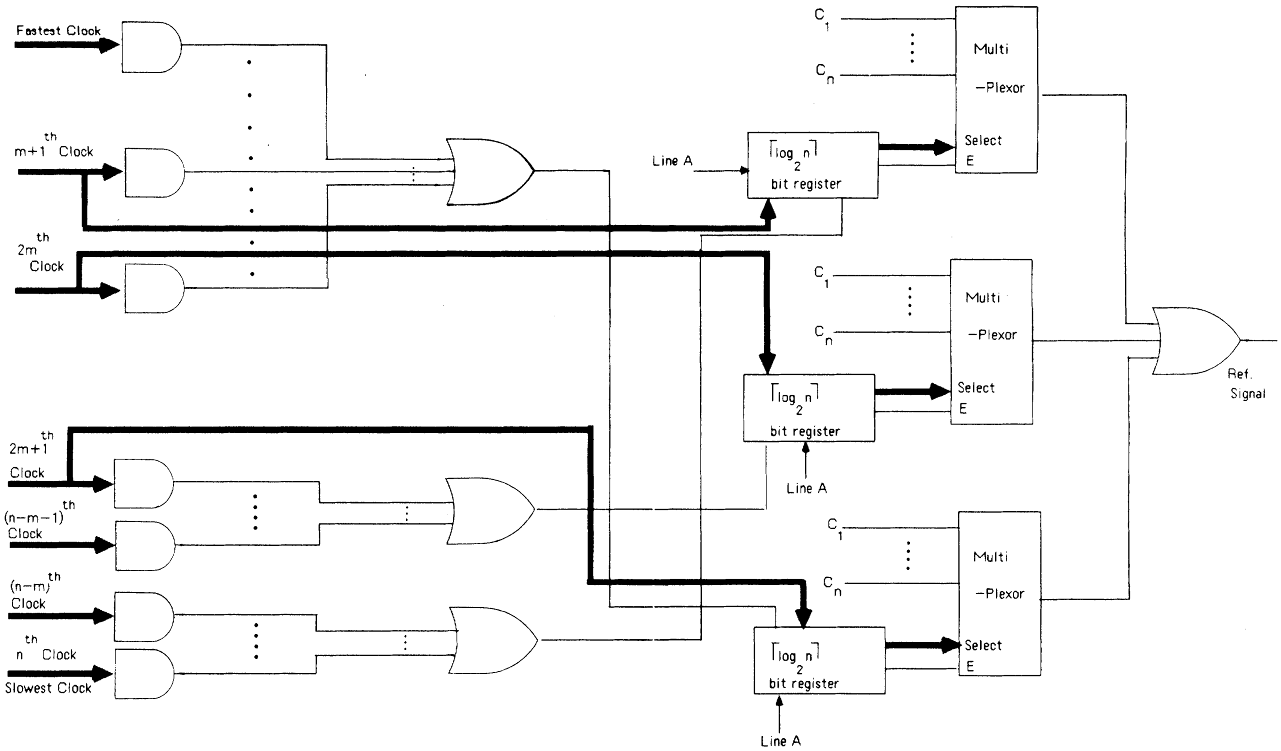


Fig. 10. Block C—reference signal generator.

B. The Complexity Analysis of the Circuit

The complexity of the entire circuit is determined by the complexity of the individual blocks. Thus, we will first examine the complexity of each of these individual blocks before determining the complexity of the entire circuit.

Note that the total number of inputs to a particular clock n is much smaller as compared to the total number of clocks in the entire network due to the interconnection strategy we developed in Section II. This point is crucial in view of the complexity of the synchronizing circuitry for large multiprocessor systems.

First, consider the complexity of Block A. For n inputs, Block A has n registers, one counter, n D-type FF's and a high-frequency clock. Similarly, Block C requires n AND gates, a few OR gates, three multiplexers and three small registers. Therefore, the complexity required by Blocks A and C together is of $O(n)$. On the other hand, the complexity of Block B grows as $O(n^2)$. This is because to sort n inputs using only two and four input sorters we require $n^2/16$ 4-input sorters and $g(n)$ 2-input sorters where $g(n)$ satisfies the equation $g(2n) = 4g(n) + 2n$, with $g(2) = 0$. This makes the complexity of the entire circuit $O(n^2)$.

C. Merits and Demerits of the Synchronizing Circuit

The best hardware implementation proposed so far grows as $O(n^m)$ in gate complexity where m is the required fault tolerance [4]. By using the same logic as in the previous section, it might be possible to design a single chip with a large number of gates (of the order of 3000 gates for 16 inputs, $m = 5$) and build a synchronizing circuitry using only about five chips. Most of these large number of gates are required for designing the combinational logic to detect whether more than m clocks differ substantially from a particular clock. A single chip for this purpose will be too dedicated to be useful for any other applications. On the other hand, single chip sorters will come in very handy in designing many database systems. Thus, it will be more economical to have a single chip sorter as compared to a single chip "greater than m detector." However, if we can afford to have such a dedicated chip, then it will be better to use the circuit in [4].

Another major disadvantage with the circuit in [4] is that it is not modular in nature. For example, suppose we design a single chip "greater than m detector" for 8 inputs. It is not possible to use it to design a 16-input synchronizing circuitry. This means that we will need another completely different dedicated chip to design a 16-input synchronizing circuitry. This will greatly increase the hardware cost, and hence make it impractical.

Since the maximum resolution between the arrival of any two ticks is determined by the frequency of the high-frequency clock (C_{hf}) in Block A, our synchronizing capability is limited only by that frequency. If the frequency of that clock is sufficiently high, then we can get a very tight synchronization. For example, if we were to use a 33 MHz clock as our high-frequency clock, then we are limited to a maximum skew of about 30 ns in this circuit. This is a very tight synchronization, and is not a limitation at all. Consequently, this circuit achieves all the desired features of a good clock synchronizing circuitry at a far less complexity.

VI. Conclusions

The new algorithm proposed in this paper is a hardware synchronization algorithm that can be used to synchronize all the clocks in a multiprocessor system of any arbitrary size (small or large). It provides an optimum tradeoff between the total number of interconnections and the fault tolerance of the system, while maintaining the symmetry of the network.

Given any desired fault tolerance specification, this algorithm can be used to determine the network architecture that uses a near minimal number of interconnections as well as a clock network very similar to the processor network. As shown in Section IV-B, the use of this algorithm could reduce the total number of interconnections by 80 percent. This might lead one to believe that this algorithm would result in a network with lesser synchronizing capabilities. As shown in Section III, however, this drastic reduction in the total number of interconnections causes little or no difference in the synchronizing capabilities. Consequently, it has a high potential for synchronizing large multiprocessor systems.

Acknowledgment

The authors wish to thank C. M. Krishna, University of Massachusetts, and the anonymous referees for their comments on this paper.

References

[1] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *J. Ass. Comput. Mach.*, vol. 32, pp. 52-78, Jan. 1985.

[2] T. B. Smith and J. H. Lala, "Development and evaluation of a fault-tolerant multiprocessor (FTMP) computer volume I: FTMP principles of operation," NASA, Contractor Rep. 166071, May 1983.

[3] C. M. Krishna, K. G. Shin, and R. W. Butler, "Ensuring fault tolerance of phase-locked clocks," *IEEE Trans. Comput.*, vol. C-34, pp. 752-756, Aug. 1985.

[4] J. L. W. Kessels, "Two designs of a fault-tolerant clocking system," *IEEE Trans. Comput.*, vol. C-33, Oct. 1984.

[5] C. M. Krishna, K. G. Shin, and R. W. Butler, "Synchronization and fault-masking in redundant real time systems," *Dig. Papers, FTCS-14*, 1984, pp. 152-157.

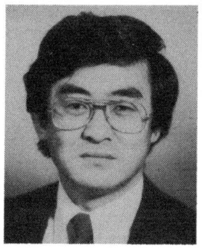
[6] D. Davies and J. F. Wakerly, "Synchronization and matching in redundant systems," *IEEE Trans. Comput.*, vol. C-27, pp. 531-539, June 1978.

[7] T. B. Smith, "Fault-tolerant clocking system," *Dig. Papers, FTCS-11*, 1981, pp. 262-264.

[8] K. G. Shin, C. M. Krishna, and Y. H. Lee, "A unified method for evaluating real-time controllers and its application," *IEEE Trans. Automat. Contr.*, vol. AC-30, pp. 357-366, Apr. 1985.

[9] D. M. Himmelblau, *Applied Non-linear Programming.* New York: McGraw-Hill, 1972, pp. 274-292.

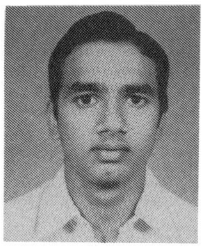
[10] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, pp. 153-161, Feb. 1971.



Kang G. Shin (S'75-M'78-SM'83) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and both the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1970 to 1972 he served in the Korean Army as an ROTC officer and from 1972 to 1974 he was on the research staff of the Korea Institute of Science and Technology, Seoul, working on the design of VHF/UHF communication systems. From 1974 to 1978 he was a Teaching/Research Assistant and then an Instructor in the School of Electrical Engineering, Cornell University. From 1978 to 1982 he was an Assistant Professor at Rensselaer Polytechnic Institute, Troy, NY. He was also a Visiting Scientist at the U.S. Air Force Flight Dynamics Laboratory in summer 1979 and at Bell Laboratories, Holmdel, NJ, in summer 1980 where his work was concerned with distributed airborne computing and cache memory architecture, respectively. He also taught short courses for the IBM Computer Science Series in the area of computer architecture. Since September 1982, he has been with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, where he is currently an Associate Professor. His current teaching and research interests are in the areas of distributed and fault-tolerant computing, computer architecture, and robotics and automation.

Dr. Shin is a member of the Association for Computing Machinery, Sigma Xi, and Phi Kappa Phi.



P. Ramanathan (S'84) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1984.

Since then he has been a Research Assistant at the University of Michigan, Ann Arbor, where he received the Masters degree in computer engineering in 1986 and is currently pursuing the Ph.D. degree. His research interests include fault modeling, distributed systems and high-performance architectures.