Optimal Fault-Secure Scheduling

JIE WU, EDUARDO B. FERNANDEZ AND DONGLAI DAI

Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431, USA
Email: jie@cse.fau.edu

We consider here two basic fault-secure scheduling problems for multiprocessor systems. First, given the number of processors in the system and a set of computational tasks of unit length expressed as a complete binary tree, a scheduling algorithm is proposed such that the total execution time is a minimum and no undetected single error result will be delivered. Second, given a deadline and a computation tree, another algorithm is given which generates a fault-secure scheduling using a minimum number of processors. We show that two previous approaches are special cases of these algorithms. We also discuss the way to modify our scheduling to ensure a given fault latency requirement. Finally, extensions that cover multiple errors, non-unit length tasks and computation graphs of arbitrary binary trees are discussed.

Received September 17, 1996; revised June 2, 1998

1. INTRODUCTION

The notion that parallel and distributed computer systems [1] could function as powerful general purpose computing facilities has existed for quite some time. One of the reasons for this is that they permit shorter execution times of applications compared to uniprocessor systems with the same technology. It is clear that they will be the key technique for the next generation of computer systems for high-performance applications [2]. Computer systems consisting of thousands of processors are not a theoretical topic anymore; however, for years, software has been the main obstacle blocking applications for taking full advantage of such systems. Decomposing, either by the application software designer or by system software, a large task into many small concurrently executable segments (or subtasks) and assigning these segments to different processors to optimize either time or number of processors is a major challenge.

A large amount of research on scheduling for multiprocessor systems has been conducted [3, 4, 5, 6, 7, 8, 9, 10]. In a multiprocessor system, when the number of processors increases, the probability of faulty processors also increases [11]. There is a need to apply appropriate fault-tolerance techniques to improve the system reliability [12] and dependability [13]. This paper is concerned with fault-secure scheduling based on fault-secure computation [14, 15]. In a fault-secure computation, the output of a system is guaranteed to be either correct or tagged as incorrect. We assume here that fault-secure computation is obtained by the use of duplication and comparison. A fault-secure schedule has the capability of detecting errors but not of correcting them; fault-tolerant scheduling [16] has also the capability of correcting the detected errors. However, fault-tolerant scheduling uses more sophisticated and more costly techniques such as triple modular redundancy (TMR) [17] and roll-forward recovery [18, 19].

The concept of fault security was first introduced in logic circuit design [20]. Banerjee and Abraham [14] applied this idea to scheduling. In their approach, each node in the computation graph (which corresponds to a task) is duplicated and their results are compared. The type of fault controlled is restricted to a single hardware fault. Since many time slots of processors are idle as a result of a conventional scheduling, they suggested the use of these idle time slots for duplicates. Comparison of the results of the duplicates achieves the fault-secure property without affecting significantly the completion time of the computation graph.

More recently, Gu *et al.* [15] further developed the idea of Banerjee and Abraham by introducing the concept of *k-fault-secure scheduling*. In such a schedule, for every fault pattern of size at most k, the output of a system is guaranteed to be either correct or tagged as incorrect. They considered schemes for some special types of computation trees and they showed that some well known parallel computation paradigms have binary trees as computation task graphs. They reduced the number of processors used in the fault-secure schedules of [14] by a factor of two or four without significantly delaying the total execution time.

However, in both [14] and [15], fault-secure scheduling algorithms produce schedules under the assumption that the number of processors is unlimited, i.e. as many processors as needed are available. This assumption is not valid for many parallel systems, either because there is a limited number of processors or because several computation trees need to be accommodated.

In this paper, we study the following two fault-secure scheduling problems, where the computation graph under consideration is a complete binary tree with unit length tasks, unit length comparators and with negligible interprocessor communication times.

1. Given the number of processors in the system, the tasks are scheduled such that the total execution time is minimized and the resultant schedule is fault-secure. For convenience, this number is restricted to 2^h, where h is a positive integer.
2. Given a deadline (not smaller than the lower bound for conventional non-fault-tolerant scheduling), tasks are scheduled on a minimum number of processors such that they can still complete within the given time limit and the resultant schedule is fault-secure.

We first find bounds for the above two optimization problems and then determine the fault-secure schedules that achieve these bounds. Given a number of processors $m = 2^h$ and a computation tree with depth p, when a particular constraint on p and h is satisfied, our fault-secure scheduling algorithm processes the computation tree taking the same minimum time as a non-fault-tolerant schedule (except one additional time unit for testing), i.e. this result matches the best result that a conventional scheduling could obtain. When this constraint on p and h does not hold, our fault-secure scheduling algorithm generates no idle slots and the result is the best a fault-secure scheduling could obtain. Similar results are also derived on the minimum number of processors required to perform a fault-secure schedule under a given time limit. We also show that the scheduling algorithms in [14] and [15] are special cases of our algorithms. Another salient feature of our approach is that there is no special requirement for the comparators, which are assigned to regular processors.

We assume that an output of a computation graph is delayed until it is checked to be correct. In our case, there is one output which is the root node of the application tree. However, in some applications such as real-time systems, an output has to be generated within a specific time frame; that is, the comparison should be done before a given deadline. A similar issue is fault latency, which is measured by the time between the occurrence of faults and the detection of these faults. In this paper, we provide general guidelines to modify the proposed scheme to satisfy a given fault latency requirement. Extensions are also discussed to cover multiple errors, non-unit tasks and computation graphs of arbitrary binary trees.

Some other approaches have been proposed in recent years [16, 21, 22, 23, 24], which use fault-tolerant techniques such as recovery blocks and backup processors. These attempt to protect against software faults or have different objectives from ours. Task duplication or replication has also been used in scheduling for purposes other than fault tolerance: in [7, 25, 26] to reduce the effect of interprocessor communication times; in [27] to consider the effect of the variability of task execution times.

Section 2 discusses background and introduces some notation. Section 3 considers fault-secure scheduling for a system with a fixed number of processors. Section 4 describes fault-secure scheduling under time constraints. Section 5 evaluates the proposed scheduling algorithms and compares our approach to [14] and [15]. Section 6 discusses extensions of the proposed approach to cover multiple faults, non-unit task lengths and computational graphs that are arbitrary binary trees. A general guideline for modifying the proposed scheme to meet a given fault latency requirement is also given in that section. Section 7 presents conclusions and some ideas for future work, especially the use of trade-offs in scheduling length, number of processors used and number of faults in the system.

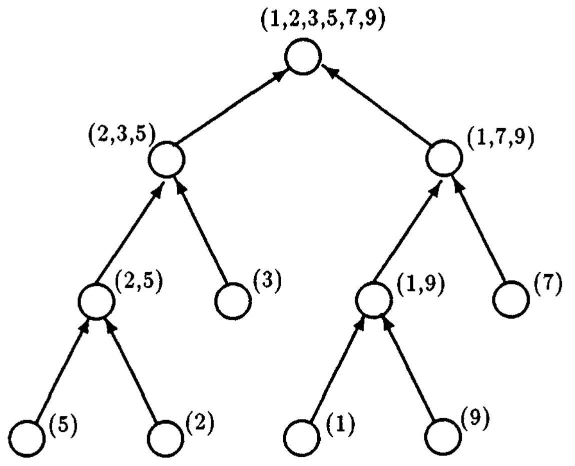


FIGURE 1. Example of parallel merge sorting.

2. BASIC FAULT-SECURE SCHEDULES

Following [14, 15], we assume a multiprocessor system consisting of a set of processors. An error propagates from one processor to another only through regular communication channels. We also assume that all the processors have identical computing power and that a processor can perform any operation of a computation, namely, a subtask or a test, within a unit of time. Similar to [14] and [15] we concentrate in this paper on complete binary trees with unit task nodes. In Section 6, we will discuss how to relax this restriction.

A computation graph G is a directed acyclic graph (DAG) in which a node N corresponds to a process or task and a directed edge (N_i, N_j) connects two nodes N_i and N_j. If there is a directed path (sequence of adjacent arcs) from N_i to N_j, we say that there is a precedence constraint between N_i and N_j. Figure 1 shows an example of a parallel version of merge sorting expressed as a tree structure. Each node in the tree corresponds to a task of a unit-time duration. Each task receives two sorted lists from lower-level tasks and produces a merged sorted list.

A schedule is typically represented using a Gantt chart, which is a two-dimensional array, with one row for each processor and one column for each time slot. Given a computation tree G and a multiprocessor system, a schedule σ maps nodes in G to entries of processor execution time, where an *entry* is a crosspoint of a column and a row of a Gantt chart; that is, each time slot has m entries where m is the number of processors. More specifically, $\sigma(N) = (P, i)$, where N represents a node in G, P is a processor and i is a unit-time slot in the execution of tasks by P.

PROPERTY. Let $G(p)$ be a computation tree with depth p; the minimum total execution time of a schedule for $G(p)$ is p.

We assume that processor faults manifest themselves as errors that occur in one or more time slots and they can be either permanent or transient. A fault-free processor P always produces the correct output if all the inputs for an operation on P are correct; it may or may not produce erroneous output if some input is erroneous.

DEFINITION. *A fault-secure schedule either produces correct outputs or tags these outputs as incorrect.*

A fault-secure system never delivers an erroneous output, although it might misjudge a correct output to be an erroneous one. A *k-fault-secure system* is a system which has the fault-secure property for any k faults in the system within a specified time period. In this paper, we concentrate on 1-fault-secure schedules, but the general case of k-fault-secure schedules is briefly discussed in Section 6.

Our approach to achieve the fault-secure property is to duplicate the execution of a task and then to compare their results. Our scheduling algorithm does this by duplicating each node and by carrying out binary equality tests on two versions. Fault detection occurs when some test reports 'not equal'. If any test reports a 'not equal' we discard this output and the system is still safe. It has been shown [15] that to ensure 1-fault-security at least two-version computation is necessary. However, depending on how versions are arranged, a comparator that compares two versions of a node may or may not be necessary. Therefore, the key issue here is how to arrange tests and duplicates. Let us start with the test arrangement. Clearly two versions of a node should be assigned to two different processors to ensure a fault-secure computation. Suppose we want to test node N and its duplicate N'; we need to decide where to locate the comparator: it could be located at one of the two processors or at a third processor. When the comparator is located at a third processor, the 1-fault-secure property is straightforward. In Figure 2a, if one of the two processors (P_1 or P_2) on which two versions are assigned is faulty, it affects the outcome of one version; the comparator (t) on P_3, a healthy processor, will catch the error. If P_3 is faulty then the comparator could generate any decision (pass or fail), but because both versions on the healthy processors produce correct results, the final outcome is still safe. If a comparator is located in one of the two processors on which two versions are assigned then 1-fault-security cannot be assured. Suppose the processor on which the comparator is assigned fails and the comparator passes the test, then the version on this faulty processor would be considered correct. To avoid this situation, two versions of the comparator (t and t' in Figure 2b) should be used and assigned to different processors. To choose between Figures 2a or 2b one should note that the configuration in Figure 2a saves one test; however, the test has to be run on a third processor.

Next we examine the possibility of reducing the total number of comparators used. Suppose that N_1 is the parent node of N_2 in the computation tree, then a direct test of the two versions of N_2 can be avoided using the configuration shown in Figure 3a. However, the output of each of the N_2s can be used as input for only one version of N_1. When a direct test is used for N_2, as shown in Figure 3b, the output from one version of N_2, say N_2, can be used as the input to both versions of N_1. In this case, there is more freedom to assign the other version N_2' to a time slot, even after N_2, N_1 and t. As we will see later, such freedom might be vital to ensure a minimum execution time. Although N_2 could be faulty and contaminate both versions of N_1, its failure will eventually be caught by a comparator. In general, in order to reduce fault latency in the latter case, N_2' should be assigned as early as possible without increasing the total execution time (this issue will be discussed in detail in Section 6). Therefore, the key in a fault-secure schedule is the selection of different configurations of the type shown in Figures 2 and 3.

Extending the idea used in Figure 2 by replacing each node by a subtree or a complete computation tree, we can derive the following scheme which uses only one or two comparators for the entire computation tree: replace nodes N and N' in Figure 2a by two versions of a complete computation tree G and each processor (except the third processor P_3) by a set of processors. We then derive two fault-secure schedules (one uses one comparator with one additional processor and the other uses two comparators but no additional processors) that use a minimum number of time slots and minimum possible time, i.e. the depth of the tree plus one. This idea results in our basic fault-secure schedule (BFSS) as will be discussed later.

We first define the concept of level of the nodes in the computation tree. Basically, the root node of $G(p)$ is assigned a level p (p will be the maximum level and all the other levels are smaller) and any node that is i distance away from the root is assigned a level $p - i$. The node set in G can be partitioned into p subsets: $L_1, L_2, \ldots, L_p$ (called L-precedence partitions). Each node is denoted as N_{ij} where i is the level at which this node is located and j is the label assigned within level i. Figure 4 shows the L-precedence partitions for the example of Figure 1. In Figure 4, each node is represented by N_{ij} where i is the level of this node and j is the label within the level (counting from the left to right).

PROPOSITION. *When $h \geq p$, the minimum total execution time of a fault-secure schedule for $G(p)$ is $p + 1$.*

We start with a basic fault-secure scheduling method that achieves the bound shown in Theorem 1 without constraints on the number of processors used. A similar scheme was presented in [14] and [15]. However, it was not presented in an algorithmic format. In the next section we extend this simple scheduling by adding constraints on either time or on the number of processors we are allowed to use.

In the basic fault-secure schedule (BFSS), the node set of $G(p)$ is partitioned into a set of precedence partitions $\{L_1, L_2, \ldots, L_p\}$. The processor set is partitioned into two sets, P_{up} and P_{down}, of equal size. The nodes in $G(p)$ are

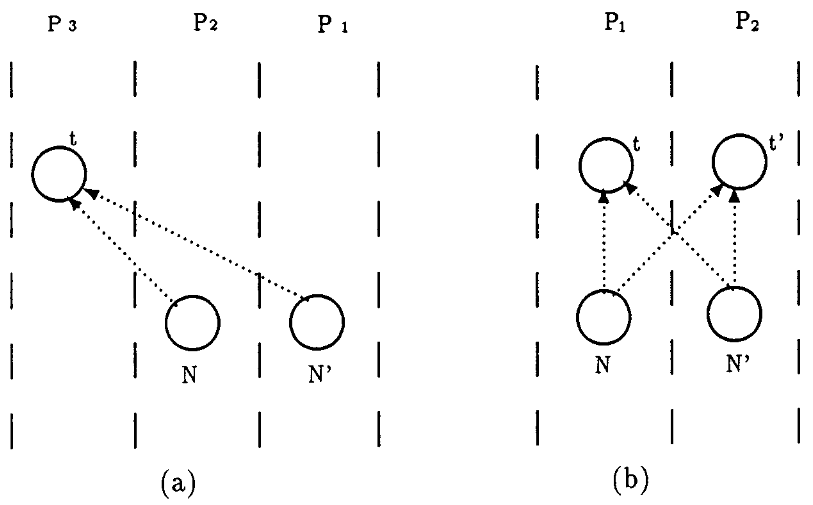


FIGURE 2. The assignment of the comparator (a) to a third processor and (b) to one of two processors used.

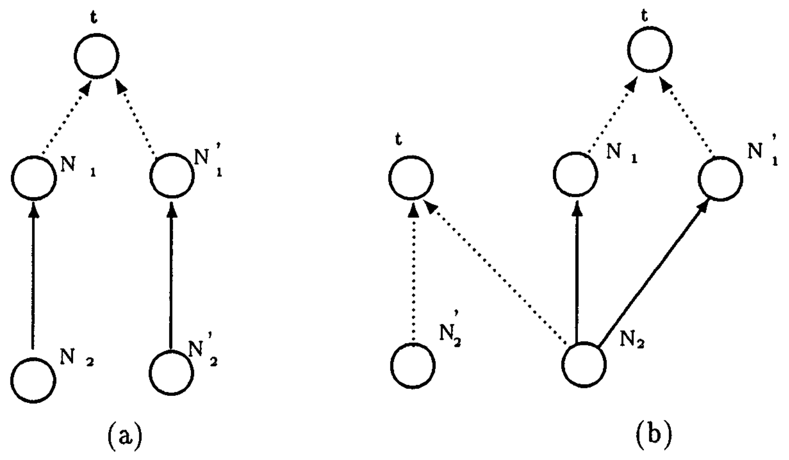


FIGURE 3. Two possible fault-secure configurations of N_2 and N_1.

assigned level by level to entries of each time slot of P_{up} in the Gantt chart; that is, nodes in level i are assigned to time slot i (we assume time slots start from 1). The duplicated node set is also assigned in the same way into P_{down}. When two comparators are used to check two versions of the root node, these two tests can be assigned to any two processors in P_{up} or P_{down} at time slot $p+1$ (the last time slot). When one comparator is used, the test has to be assigned to a processor not in P_{up} or P_{down}. Clearly, the minimum number of processors used is $|P_{\text{up}}|+|P_{\text{down}}| = 2*\max\{|L_i|\}$, where $1 \le i \le p$, and two comparators are used. One additional processor is required when one comparator is used. In summary, the algorithm can be expressed as:

```
ALGORITHM Basic_Fault_Secure_Schedule (BFSS)
(1) partition the node set of $G(p)$ into $\{L_1, L_2, \ldots, L_p\}$;
(2) divide the processor set into two equal-sized
    partitions, $P_{\text{up}}$ and $P_{\text{down}}$;
(3) assign nodes in $L_i$ to entries of slot $i$ of processors in
    $P_{\text{up}}$;
(4) assign duplicates of nodes in $L_i$ to entries of slot $i$ of
    processors in $P_{\text{down}}$;
(5) case one comparator $\rightarrow$
        assign the comparator to a processor not in $P_{\text{up}}$
        or $P_{\text{down}}$ at time slot $p+1$
    two comparators $\rightarrow$
        assign two comparators to any two processors in
        $P_{\text{up}}$ or $P_{\text{down}}$ at time slot $p+1$
    end case
```

In the computation tree shown in Figure 4, $\max\{L_1, L_2, L_3, L_4\} = 4$. Figure 5 shows a schedule produced by the BFSS algorithm for the graph of Figure 4 where $P_{\text{up}} = \{P_1, P_2, P_3, P_4\}$ and $P_{\text{down}} = \{P_5, P_6, P_7, P_8\}$ and two comparators are used. Each number in the first line corresponds to time slot i. If N_j and N'_j are two versions of the same node, we use $t(N_j, N'_j)$ to represent the equality test on N_j. If the test itself has two versions, we use $t(N_j, N'_j)$ and $t'(N_j, N'_j)$ to represent them. $N^*_k(N^*_i, N^*_j)$ represents the fact that the computation of N^*_k requires inputs from N^*_i and N^*_j.

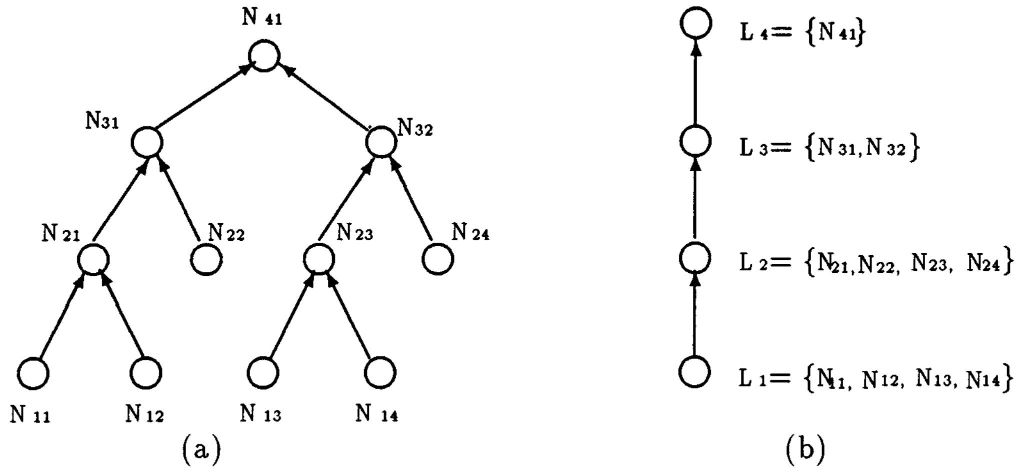


FIGURE 4. Example of L-precedence partitions.

	1	2	3	4	5
P_1	N_{11}	$N_{21}(N_{11}, N_{12})$	$N_{31}(N_{21}, N_{22})$		
P_2	N_{12}	N_{22}	$N_{32}(N_{23}, N_{24})$		
P_3	N_{13}	$N_{23}(N_{13}, N_{14})$		$N_{41}(N_{31}, N_{32})$	
P_4	N_{14}	N_{24}			$t(N_{41}, N'_{41})$
P_5	N'_{11}	$N'_{21}(N'_{11}, N'_{12})$	$N'_{31}(N'_{21}, N'_{22})$		
P_6	N'_{12}	N'_{22}	$N'_{32}(N'_{23}, N'_{24})$		
P_7	N'_{13}	$N'_{23}(N'_{13}, N'_{14})$		$N'_{41}(N'_{31}, N'_{32})$	
P_8	N'_{14}	N'_{24}			$t'(N'_{41}, N'_{41})$

FIGURE 5. Fault-secure scheduling for the computation graph of Figure 1 by the basic fault-secure schedule.

3. FAULT-SECURE SCHEDULING WITH A FIXED NUMBER OF PROCESSORS

We now study bounds on time for a fault-secure schedule when the number of processors is fixed. We assume that the number of processors m is always a power of 2, say $m = 2^h$, where h is a positive integer. When $h > p$, there are enough processors to carry out the basic fault-secure scheduling and the processing time is bounded by $p + 1$. When $h \leq p$, there are not enough processors to execute according to the BFSS algorithm and we need to develop other fault-secure scheduling algorithms. Since at least two processors are needed to perform a fault-secure computation, h is always greater than zero.

For a non-fault-secure scheduling of an arbitrary computation tree G, Hu [28] found the following lower bound.

LEMMA 1. [28] *Given m processors, the minimum time T required to process a tree $G(p)$ of unit tasks is bounded as follows.*

Let

$$q = \max_{1 \leq x \leq p} \left\{ \frac{1}{m} \sum_{j=1}^{x} |L_j| - x \right\} \tag{1}$$

then

$$T \geq p + \lceil q \rceil. \tag{2}$$

LEMMA 2. *Given $m = 2^h$ processors and a complete binary tree $G(p)$ with $h \leq p$, then $T \geq 2^{p-h} + h - 1$.*

Proof. For the complete tree the sequence in (1) becomes: when $x = 1$,

$$2^{p-1}\frac{1}{2^h} - 1 = 2^{p-h-1} - 1,$$

when $x = 2$,

$$[2^{p-1} + 2^{p-2}]\frac{1}{2^h} - 2 = (2^{p-h-1} - 1) + (2^{p-h-2} - 1),$$

. . . ,
when $x = p - h$,

$$[2^{p-1} + 2^{p-2} + \ldots + 2^{h+1} + 2^h]\frac{1}{2^h} - (p - h)$$
$$= (2^{p-h-1} - 1) + (2^{p-h-2} - 1) + \ldots + (2^1 - 1) + (2^0 - 1),$$

when $x = p - h + 1$,

$$[2^{p-1} + 2^{p-2} + \ldots + 2^{h+1} + 2^h + 2^{h-1}]\frac{1}{2^h} - (p - h + 1)$$
$$= (2^{p-h-1} - 1) + (2^{p-h-2} - 1) + \ldots + (2^1 - 1) + (2^0 - 1) + (2^{-1} - 1),$$

. . . ,
when $x = p$,

$$[2^{p-1} + 2^{p-2} + \ldots + 2^{h+1} + 2^h + 2^{h-1} + \ldots + 2^0]\frac{1}{2^h} - p$$
$$= (2^{p-h-1} - 1) + (2^{p-h-2} - 1) + \ldots + (2^1 - 1) + (2^0 - 1) + (2^{-1} - 1) + \ldots + (2^{-h} - 1).$$

Since $h \leq p$, the maximum item of the above sequence is at $x = p - h$.

From (1), we have

$$q = [2^{p-1} + 2^{p-2} + \ldots + 2^{h+1} + 2^h]\frac{1}{2^h} - (p - h)$$
$$= \frac{2^{p-h-1} * 2 - 1}{2 - 1} - (p - h) = 2^{p-h} - p + h - 1.$$

From (2), we have

$$T \geq p + \lceil q \rceil = 2^{p-h} + h - 1. \qquad \square$$

Theorem 1. *Given $m = 2^h$ processors and a complete binary tree $G(p)$ with $h \leq p$, the minimum time T required for fault-secure scheduling is larger than or equal to $2^{p-h} + h$.*

Theorem 1 is directly derived from Lemma 2 by adding one to the bound in Lemma 2, i.e. an extra time slot is required in fault-secure schedules for the final test. Note that Hu's bound is for non-fault-secure schedules; that is, schedules where tasks are not duplicated. Therefore, if a fault-secure schedule matches this bound (with one more step), this schedule is optimal. The following theorem reveals another bound based on an ideal fault-secure scheduling that does not create any idle entries in a Gantt chart.

Theorem 2. *Given $m = 2^h$ processors and a complete binary tree $G(p)$, let T be the total execution time of a fault-secure schedule. If $h \leq p$, then*

$$T \geq \begin{cases} 2^{p-h+1} & h = 1 \\ 2^{p-h+1} + 1 & h \neq 1. \end{cases} \qquad (3)$$

Proof. Since the number of nodes in $G(p)$ is $2^p - 1$, the total load (computation cost) for a fault-secure two-version computation is $2^{p+1} - 2$. If we ignore the precedence constraints in $G(p)$, the number of time slots required for the computations is

$$\left\lceil \frac{2^{p+1} - 2}{2^h} \right\rceil = \lceil 2^{p-h+1} - 2^{-(h-1)} \rceil$$
$$= \begin{cases} 2^{p-h+1} - 1 & h = 1 \\ 2^{p-h+1} & h \neq 1. \end{cases}$$

This will obviously serve as a lower bound on this part of the execution time. We need another time slot for the final test, that is

$$T \geq \begin{cases} 2^{p-h+1} & h = 1 \\ 2^{p-h+1} + 1 & h \neq 1. \end{cases} \qquad \square$$

Combining the results from Theorems 1 and 2, we have the following bounds on time for a fault-secure computation.

Theorem 3. *Given $m = 2^h$ processors and a complete binary computation tree $G(p)$, the minimum time T required to process a fault-secure computation of G on m processors is bounded as follows:*

$$T = \begin{cases} p + 1 & h \geq p \\ 2^{p-h} + h & 1 < h < p \wedge h + \log(h-1) \geq p \\ 2^{p-h+1} + 1 & 1 < h < p \wedge h + \log(h-1) < p \\ 2^p & 1 = h < p. \end{cases} \qquad (4)$$

Proof. When $h > p$, from Theorem 1 we have that $T = p + 1$. When $h = p$, results from Theorems 1, 2, and 3 can be used; we have $T = \max\{p + 1, p + 1, 2 + 1$ (when $p \neq 1$) or $1 + 1$ (when $p = 1$)$\} = p + 1$. Therefore, when $h \geq p, T = p + 1$. When $h < p$ and $h = 1$, we have $T = \max\{2^{p-1} + 1, 2^p\} = 2^p$ based on Theorems 2 and 3. When $1 < h < p$, again results from both Theorems 2 and 3 can be used. In this case, the higher bound out of the two is selected. We first determine the point where Theorems 2 and 3 generate the same bound. Suppose $2^{p-h} + h - 1 = 2^{p-h+1} + 1$, then $h + \log(h - 1) = p$. It is easy to see that when $h + \log(h - 1) > p$ the bound from Theorem 2 is higher than the bound from Theorem 3 and the opposite situation is true when $h + \log(h - 1) < p$. $\square$

Based on Theorem 3, we have the following important observation. Assume that the bounds in Theorem 3 are achievable, then when $h + \log(h - 1) \geq p$ (which includes the first two bounds in Theorem 3) the corresponding fault-secure schedule processes the computation as fast as any non-fault-secure schedule where nodes in the computation are not duplicated. When $h + \log(h - 1) < p$ (which includes the last two bounds in Theorem 3) the corresponding fault-secure schedule does not generate idle entries (except entries in the last time slot used for comparison).

Our next goal is to find fault-secure schedules that meet the bounds of Theorem 4 (if such schedules exist). When

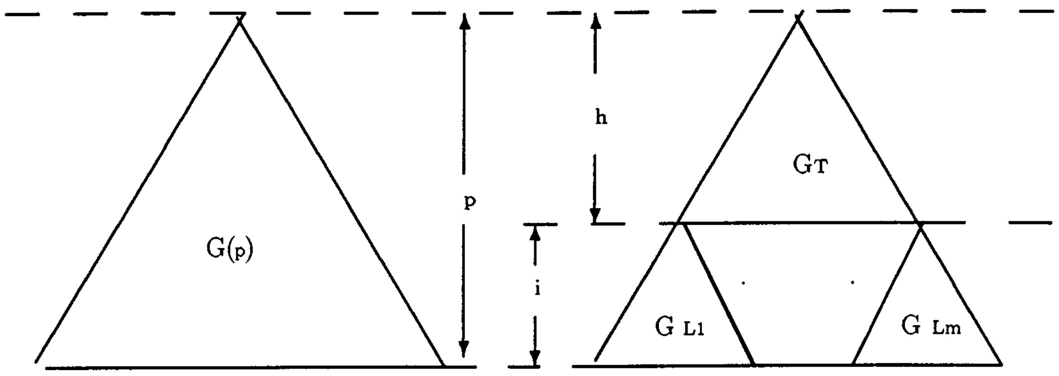


FIGURE 6. Partition of G_p into G_T and G_{L_j}.

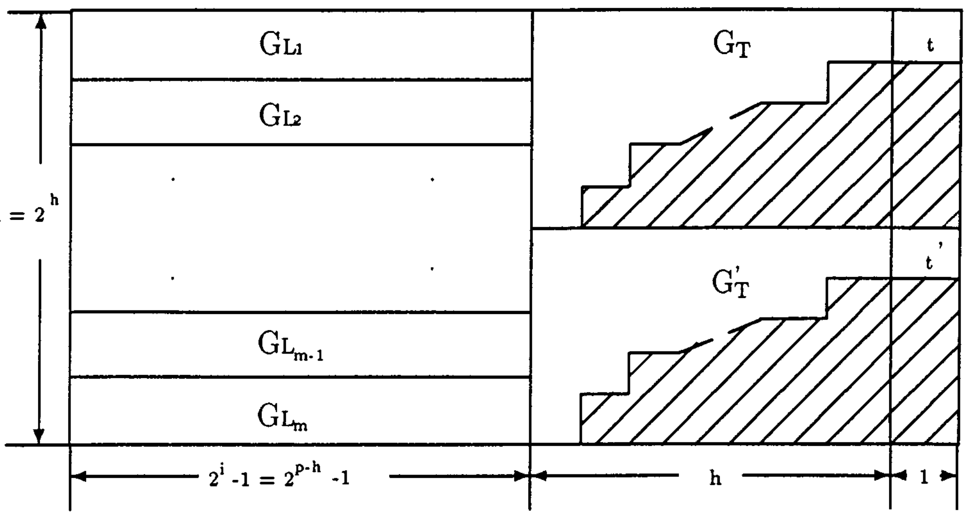


FIGURE 7. Fault-secure scheduling when $p > h$.

$h \geq p$ the basic fault-secure scheduling algorithm can be directly applied. When $1 = h < p$, i.e. there are two processors, we define a total order (or topological sorting order) of the computation tree based on the precedence order. Nodes in G are assigned to one processor based on this total order and duplicates are assigned to the other processor. Since there are $2^p - 1$ nodes in G, 2^p time slots are needed (which include the slot used for comparison). Obviously, there are no idle entries in the resultant Gantt chart.

When $1 < h < p$, there are not enough processors available to execute the simple fault-secure scheduling. Assume a complete binary tree G_T of depth h is a subtree of the given computation tree G that shares the same root node with G (see Figure 6). Let $i = p - h$, i.e. i is the number of levels in G below the subtree G_T. Nodes in the lower i levels consist of $m = 2^h$ independent complete binary trees G_{L_j} of depth i. The basic scheduling strategy is as follows. Each G_{L_j}, $1 \leq j \leq m$, is first assigned to m distinct processors. In this case, the first $2^{p-h} - 1$ time slots are used to process one version of all the G_{L_j} (see Figure 7). Then the simple fault-secure schedule is used to assign two versions of G_T using additional $h+1$ time slots. The idle entries (the shaded areas in Figure 7) generated in the scheduling of G_T, are used to assign the duplicates and tests of nodes in G_{L_j}.

Depending on the values of p and h, the idle entries may not be enough for the duplicates and the tests of nodes in G_{L_j}. Therefore, additional time slots might be needed. Let us first determine the point where there are just enough idle entries. We can achieve this objective in three steps. First, count the number of idle slots in the fault-secure schedule of the G_T part which may possibly be used in fault-secure computation for the nodes in G_{L_j}. Second, count the number of duplicates in G_{L_j} and tests of G_{L_j}. Third, arrange these computations and tests in the idle entries.

As the first step, we count the number of idle entries in the fault-secure schedule of G_T. The total number of entries in the schedule for the G_T part is $m(h+1) = 2^h(h+1)$. $2(2^h - 1) - 2$ entries are used to include the two versions of G_T and two entries for the two tests, and hence the total number of idle time slots in the schedule of G_T is

$$2^h(h+1) - 2(2^h - 1) - 2 = 2^h(h-1). \qquad (5)$$

As the second step, we count the number of entries required for the duplicates and tests of G_{L_j}, to provide the fault-secure property for the whole $G(p)$.

Since the output of one version of G_{L_j} is used as the input to both versions of G_T, at least one test is required for each subtree G_{L_j}. That is, 2^h entries are needed for the tests and obviously $2^h(2^i - 1)$ entries are used for the duplicates. Then, the total number of entries required for the duplicates and tests of G_{L_j} is

$$2^h + 2^h(2^i - 1) = 2^{h+i}. \tag{6}$$

Equating expression (6) to (5), we get $2^{h+i} = 2^h(h-1)$, then $i = \log(h-1)$.

When $i \le \log(h-1)$, no additional entries are used and the total time is

$$(2^i - 1) + h + 1 = 2^i + h = 2^{p-h} + h$$

which achieves the bound (when $i \le \log(h-1)$) of Theorem 4. When $i > \log(h-1)$, the additional time needed can be calculated as follows:

$$\frac{2^{h+i} - 2^h(h-1)}{2^h} = 2^i - h + 1.$$

By adding the time used as shown in Figure 7 to the additional time calculated above, we have

$$(2^i + h) + (2^i - h + 1) = 2^i 2 + 1 = 2^{p-h+1} + 1$$

which achieves the bound (when $i \ge \log(h-1)$) of Theorem 4.

Notice that the above derivation gives us only a necessary condition since we have not considered the precedence constraints of the computation G_{L_j}. The sufficiency condition is obvious. We assign duplicates level by level starting from nodes in level one and tests are assigned last after all the nodes in the corresponding subtree have been assigned. The idle entries to which duplicated tasks and tests are assigned are also arranged level by level starting from level $2^i + 1$ (see Figure 7). The only constraint that needs to be followed is that the two versions of each node and the test for these two versions should be assigned to three different processors. The enforcement of this constraint is straightforward and we leave it for the reader to complete the assignment. In summary, the general fault-secure scheduling algorithm can be expressed as:

```
ALGORITHM General_Fault_Secure_Schedule (GFSS)
case
$p \le h \rightarrow$
  begin
  use algorithm Basic_Fault_Secure_Schedule (BFSS)
  end
$1 = h < p \rightarrow$    /* there are two processors */
  begin
  (1) define a total order of the computation tree based
      on the precedences;
  (2) assign nodes in $G(p)$ ($2^p - 1$ in all) to a processor
      based on the total order;
  (3) assign duplicated nodes in $G(p)$ to another processor
      based on the total order;
  (4) assign comparators to two processors at time slot $2^p$;
  end
$1 < h < p \rightarrow$
  begin
  (1) define a complete binary tree $G_T$ of depth $h$ as a
      subtree of $G(p)_j$ that shares the same root node
      with $G(p)$;
  (2) define complete binary trees $G_{L_j}$, $1 \le j \le 2^h$, as
      subtrees of $G(p)$ with their root nodes being nodes
      at the $(h+1)$th level;
  (3) assign nodes in $G_{L_j}$ to the first $2^{p-h} - 1$ time slots
      of distinct processors;
  (4) assign two versions of $G_T$ using BFSS starting from
      slots $2^{p-h}$;
  (5) if $p - h > \log(h-1)$ then
      $2^{p-h} - h + 1$ additional slots are required assign
      the second version of $G_{L_j}$ to the idle entries
      generated in the scheduling of $G_T$ subject to:
      (i)   the duplicated version meets the precedence
            constraints;
      (ii)  the comparator is assigned after the assignment
            of the two versions to be compared;
      (iii) two versions of each node and their comparator
            are assigned to three different processors
  end
end case
```

Figure 8 shows an assignment of a four-level complete binary tree to $m = 2^h = 8$ processors. Since $i = \log(h-1)$ in this case, where $i = 1$ and $h = 3$, no idle entry is generated. Clearly, this fault-secure schedule matches the bound for non-fault-tolerant schedules. Figure 9 shows another fault-secure schedule that corresponds to the case of $i = p - h \ge \log(h-1)$, where $i = 3$ and $h = 3$. Again, no idle entry is generated.

Based on the GFSS algorithm, we can easily show that given a number of processors $m = 2^h$, where h is a positive integer, for any complete binary computation tree $G(p)$, there exists a fault-secure schedule that meets the bounds defined by Theorem 3.

4. FAULT-SECURE SCHEDULING UNDER DEADLINES

There are cases where we are allowed to exceed the minimum time $p + 1$. We now consider how to determine the minimum number of processors $G(p)$ in a fault-secure computation under a given deadline $p + 1 + c$, where c is a non-negative integer.

We generated in the last section a general fault-secure scheduling algorithm that produces fault-secure schedules with minimum total execution times for a given number of processors m ($m = 2^h$). Now the problem becomes: given a $G(p)$, select a schedule that produces a fault-secure schedule of time $T \le p + 1 + c$ and uses a minimum number of processors.

First let us consider the delays, compared to the minimum

	1	2	3	4	5
P_1	N_{11}	$N_{21}(N_{11}, N_{12})$	$N_{31}(N_{21}, N_{22})$	N'_{18}	$t(N_{16}, N'_{16})$
P_2	N_{12}	$N_{22}(N_{13}, N_{14})$	$N_{32}(N_{23}, N_{24})$	N'_{11}	$t(N_{18}, N'_{18})$
P_3	N_{13}	$N_{23}(N_{15}, N_{16})$	N'_{12}	$N_{41}(N_{31}, N_{32})$	$t(N_{11}, N'_{11})$
P_4	N_{14}	$N_{24}(N_{17}, N_{18})$	N'_{13}	$t(N_{17}, N'_{17})$	$t(N_{41}, N'_{41})$
P_5	N_{15}	$N'_{21}(N_{11}, N_{12})$	$N'_{31}(N'_{21}, N'_{22})$	N'_{14}	$t(N_{12}, N'_{12})$
P_6	N_{16}	$N'_{22}(N_{13}, N_{14})$	$N'_{32}(N'_{23}, N'_{24})$	N'_{15}	$t(N_{14}, N'_{14})$
P_7	N_{17}	$N'_{23}(N_{15}, N_{16})$	N'_{16}	$N'_{41}(N'_{31}, N'_{32})$	$t(N_{15}, N'_{15})$
P_8	N_{18}	$N'_{24}(N_{17}, N_{18})$	N'_{17}	$t(N_{13}, N'_{13})$	$t'(N_{41}, N'_{41})$

FIGURE 8. An example of fault-secure scheduling.

execution time $p+1$, in the optimal results obtained in the last section,

- the delay is $D_1 = 0$ when $h \geq p$;
- the delay is $D_2 = h + 2^{p-h} - (p+1)$ when $(1 < h < p) \wedge (h + \log(h-1) \geq p)$;
- the delay is $D_3 = 2^{p-h+1} + 1 - (p+1)$ when $(1 < h < p) \wedge (h + \log(h-1) \leq p)$;
- the delay is $D_4 = 2^p - (p+1) = 2^p - p - 1$ when $1 = h < p$.

We intend to determine $m = 2^h$, where h is a minimum positive integer such that D_i, $i \leq r \leq 4$, is less than or equal to a given c. More precisely,

- let H_1 be the minimum h in a basic fault-secure schedule that satisfies
$$0 \leq c \wedge p \leq q;$$
- let H_2 be the minimum h in a fault-secure solution that satisfies
$$h+2^{p-h}-(p+1) \leq c \wedge 1 < h < p \wedge p \leq h+\log(h-1);$$
- let H_3 be the minimum h in a fault-secure solution that satisfies
$$2^{p-h+1} + 1 - (p+1) \leq c \wedge 1 < h < p \wedge h + \log(h-1) < p;$$
- let $H_4 = 1$ in a fault-secure solution that uses two processors and satisfies
$$2^p - p - 1 \leq c.$$

Then the selection of h is as follows,

$$h = \min\{H_1, H_2, H_3, H_4\}.$$

Notice that when $c = 0$, H_1 will give us the solution where $h = p$, and H_2 will give us the solution where $h = p - 1$. Taking the minimum, $h = p - 1$, i.e. $m = 2^{p-1}$ is the minimum number of processors to execute the fault-secure computation of $G(p)$ in $p + 1$ time slots. For $c > 0$ the solution set H_1 is empty. When $c \geq 2^p - p - 1$, $H_4 = 1$ is selected.

A detailed treatment of fault-secure scheduling under deadlines can be found in [29].

5. DISCUSSION AND COMPARISON

In the performance analysis of a parallel system the number of processors used and the execution time are considered as measures of quality. We define two criteria to measure the schedules developed by the proposed algorithms.

DEFINITION. *Given a computation $G(p)$, the time overhead ratio T_o is*

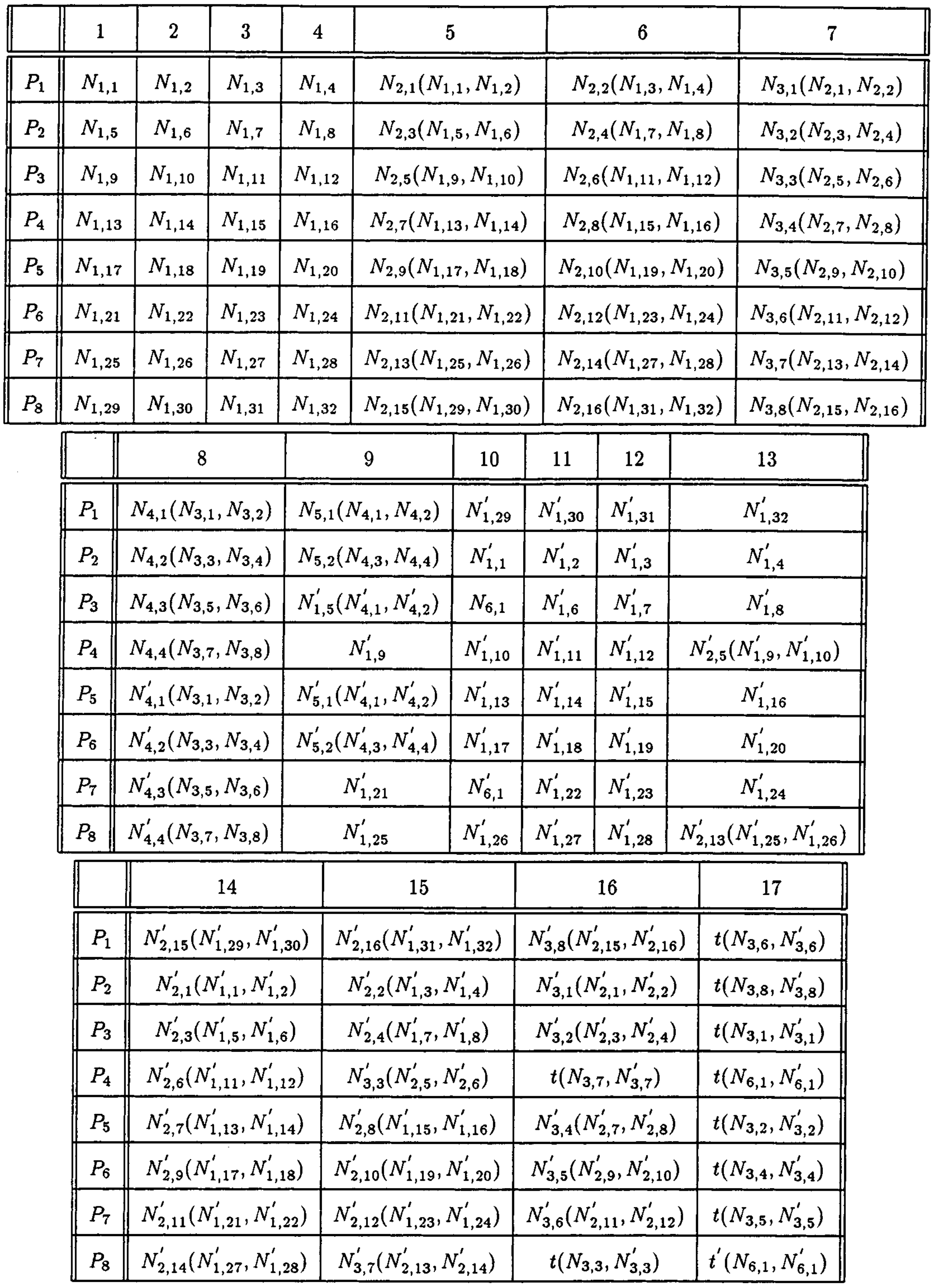
$$T_o = \frac{\textit{total time of fault secure schedules } (T_{NFS})}{\textit{total time of conventional schedules } (T_{FS})} - 1. \quad (7)$$

DEFINITION. *Given a computation tree $G(p)$, the processor utilization U_P is*

$$U_P = \frac{\textit{number of non-idle entries in schedules } (E_b)}{\textit{number of total entries in schedules } (E_t)}. \quad (8)$$

THEOREM 4. *Given $G(p)$, the time overhead ratio of fault-secure schedules produced by the general fault-secure algorithm (GFSS) is*

$$T_o = \begin{cases} 1/p & h \geq p \\ 1/(2^{p-h} + h - 1) & 1 < h < p \wedge h + \log(h-1) \geq p \\ 1 + ((3-2h)/(2^{p-h} + h - 1)) & 1 < h < p \wedge h + \log(h-1) < p \\ 1 & 1 = h < p. \end{cases} \quad (9)$$



	1	2	3	4	5	6	7
P_1	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$	$N_{1,4}$	$N_{2,1}(N_{1,1}, N_{1,2})$	$N_{2,2}(N_{1,3}, N_{1,4})$	$N_{3,1}(N_{2,1}, N_{2,2})$
P_2	$N_{1,5}$	$N_{1,6}$	$N_{1,7}$	$N_{1,8}$	$N_{2,3}(N_{1,5}, N_{1,6})$	$N_{2,4}(N_{1,7}, N_{1,8})$	$N_{3,2}(N_{2,3}, N_{2,4})$
P_3	$N_{1,9}$	$N_{1,10}$	$N_{1,11}$	$N_{1,12}$	$N_{2,5}(N_{1,9}, N_{1,10})$	$N_{2,6}(N_{1,11}, N_{1,12})$	$N_{3,3}(N_{2,5}, N_{2,6})$
P_4	$N_{1,13}$	$N_{1,14}$	$N_{1,15}$	$N_{1,16}$	$N_{2,7}(N_{1,13}, N_{1,14})$	$N_{2,8}(N_{1,15}, N_{1,16})$	$N_{3,4}(N_{2,7}, N_{2,8})$
P_5	$N_{1,17}$	$N_{1,18}$	$N_{1,19}$	$N_{1,20}$	$N_{2,9}(N_{1,17}, N_{1,18})$	$N_{2,10}(N_{1,19}, N_{1,20})$	$N_{3,5}(N_{2,9}, N_{2,10})$
P_6	$N_{1,21}$	$N_{1,22}$	$N_{1,23}$	$N_{1,24}$	$N_{2,11}(N_{1,21}, N_{1,22})$	$N_{2,12}(N_{1,23}, N_{1,24})$	$N_{3,6}(N_{2,11}, N_{2,12})$
P_7	$N_{1,25}$	$N_{1,26}$	$N_{1,27}$	$N_{1,28}$	$N_{2,13}(N_{1,25}, N_{1,26})$	$N_{2,14}(N_{1,27}, N_{1,28})$	$N_{3,7}(N_{2,13}, N_{2,14})$
P_8	$N_{1,29}$	$N_{1,30}$	$N_{1,31}$	$N_{1,32}$	$N_{2,15}(N_{1,29}, N_{1,30})$	$N_{2,16}(N_{1,31}, N_{1,32})$	$N_{3,8}(N_{2,15}, N_{2,16})$

	8	9	10	11	12	13
P_1	$N_{4,1}(N_{3,1}, N_{3,2})$	$N_{5,1}(N_{4,1}, N_{4,2})$	$N'_{1,29}$	$N'_{1,30}$	$N'_{1,31}$	$N'_{1,32}$
P_2	$N_{4,2}(N_{3,3}, N_{3,4})$	$N_{5,2}(N_{4,3}, N_{4,4})$	$N'_{1,1}$	$N'_{1,2}$	$N'_{1,3}$	$N'_{1,4}$
P_3	$N_{4,3}(N_{3,5}, N_{3,6})$	$N'_{1,5}(N'_{4,1}, N'_{4,2})$	$N_{6,1}$	$N'_{1,6}$	$N'_{1,7}$	$N'_{1,8}$
P_4	$N_{4,4}(N_{3,7}, N_{3,8})$	$N'_{1,9}$	$N'_{1,10}$	$N'_{1,11}$	$N'_{1,12}$	$N'_{2,5}(N'_{1,9}, N'_{1,10})$
P_5	$N'_{4,1}(N_{3,1}, N_{3,2})$	$N'_{5,1}(N'_{4,1}, N'_{4,2})$	$N'_{1,13}$	$N'_{1,14}$	$N'_{1,15}$	$N'_{1,16}$
P_6	$N'_{4,2}(N_{3,3}, N_{3,4})$	$N'_{5,2}(N'_{4,3}, N'_{4,4})$	$N'_{1,17}$	$N'_{1,18}$	$N'_{1,19}$	$N'_{1,20}$
P_7	$N'_{4,3}(N_{3,5}, N_{3,6})$	$N'_{1,21}$	$N'_{6,1}$	$N'_{1,22}$	$N'_{1,23}$	$N'_{1,24}$
P_8	$N'_{4,4}(N_{3,7}, N_{3,8})$	$N'_{1,25}$	$N'_{1,26}$	$N'_{1,27}$	$N'_{1,28}$	$N'_{2,13}(N'_{1,25}, N'_{1,26})$

	14	15	16	17
P_1	$N'_{2,15}(N'_{1,29}, N'_{1,30})$	$N'_{2,16}(N'_{1,31}, N'_{1,32})$	$N'_{3,8}(N'_{2,15}, N'_{2,16})$	$t(N_{3,6}, N'_{3,6})$
P_2	$N'_{2,1}(N'_{1,1}, N'_{1,2})$	$N'_{2,2}(N'_{1,3}, N'_{1,4})$	$N'_{3,1}(N'_{2,1}, N'_{2,2})$	$t(N_{3,8}, N'_{3,8})$
P_3	$N'_{2,3}(N'_{1,5}, N'_{1,6})$	$N'_{2,4}(N'_{1,7}, N'_{1,8})$	$N'_{3,2}(N'_{2,3}, N'_{2,4})$	$t(N_{3,1}, N'_{3,1})$
P_4	$N'_{2,6}(N'_{1,11}, N'_{1,12})$	$N'_{3,3}(N'_{2,5}, N'_{2,6})$	$t(N_{3,7}, N'_{3,7})$	$t(N_{6,1}, N'_{6,1})$
P_5	$N'_{2,7}(N'_{1,13}, N'_{1,14})$	$N'_{2,8}(N'_{1,15}, N'_{1,16})$	$N'_{3,4}(N'_{2,7}, N'_{2,8})$	$t(N_{3,2}, N'_{3,2})$
P_6	$N'_{2,9}(N'_{1,17}, N'_{1,18})$	$N'_{2,10}(N'_{1,19}, N'_{1,20})$	$N'_{3,5}(N'_{2,9}, N'_{2,10})$	$t(N_{3,4}, N'_{3,4})$
P_7	$N'_{2,11}(N'_{1,21}, N'_{1,22})$	$N'_{2,12}(N'_{1,23}, N'_{1,24})$	$N'_{3,6}(N'_{2,11}, N'_{2,12})$	$t(N_{3,5}, N'_{3,5})$
P_8	$N'_{2,14}(N'_{1,27}, N'_{1,28})$	$N'_{3,7}(N'_{2,13}, N'_{2,14})$	$t(N_{3,3}, N'_{3,3})$	$t'(N_{6,1}, N'_{6,1})$

FIGURE 9. An example of fault-secure scheduling.

Proof. Given a computation tree $G(p)$, and $m = 2^h$ processors in M, the total execution time of a conventional (non-fault-secure) schedule is

$$T_{NFS} = \begin{cases} p & h \geq p \\ 2^{p-h} + h - 1 & 1 < h < p \\ 2^{p-1} & 1 = h < p. \end{cases}$$

Similarly, the total execution time of a fault-tolerant schedule generated by the general fault-secure scheduling algorithm (GFSS) is

$$T_{FS} = \begin{cases} p + 1 & h \geq p \\ 2^{p-h} + h & 1 < h < p \wedge h + \log(h-1) \geq p \\ 2^{p-h+1} + 1 & 1 < h < p \wedge h + \log(h-1) \leq p \\ 2^p & 1 = h < p. \end{cases}$$

The theorem follows directly from the definition of T_0. □

Based on Theorem 4, an insignificant amount of overhead is generated in the first two cases, i.e. when $h \geq p$ or $1 < h < p \wedge h + \log(h-1) \geq p$. See Figure 10.

THEOREM 5. *Given $G(p)$, the processor utilization of a fault-secure schedule produced by our general fault-secure algorithm is*

$$U_P = \begin{cases} 1/2^{h-p-1}(p+1) & h > p \\ (2^{p-h+1}+1)/(2^{p-h}+h) & 1 < h < p \wedge h \\ & \quad + \log(h-1) \geq p \\ 1 & 1 < h < p \wedge h \\ & \quad + \log(h-1) \leq p \\ 1 & 1 = h < p. \end{cases} \tag{10}$$

Proof. Given a computation tree $G(p)$ and $m = 2^h$ processors the total number of slots (E_t) in a fault-secure schedule generated by algorithm GFSS is

$$\begin{aligned} E_t &= m * T \\ &= \begin{cases} 2^h(p+1) & h \geq p \\ 2^h(2^{p-h}+h) & 1 < h < p \wedge h \\ & \quad + \log(h-1) \geq p \\ 2^h(2^{p-h+1}+1) & 1 < h < p \wedge h \\ & \quad + \log(h-1) \leq p \\ 2^{p+1} & 1 = h < p. \end{cases} \end{aligned}$$

Similarly, the total number of non-idle (or busy) entries in a fault-secure schedule generated by algorithm GFSS is

$$E_b = \begin{cases} 2^{p+1} & h \geq p \\ 2^{p+1} + 2^h & 1 < h < p \\ 2^{p+1} & 1 = h < p. \end{cases}$$

The theorem follows directly from the definition of processor utilization $U_p = E_b/E_t$ (see Figure 11). □

It is easy to see that the fault-secure scheduling algorithms in [14] and [15] are special cases of our algorithm when $h = p$ and $h = p - 1$.

6. EXTENSIONS

6.1. Fault latency

Fault latency is defined as the time between the occurrence of a fault and the detection of this fault. In our discussion on fault-secure scheduling, we do not consider the fault latency issue, i.e. the duplicate and the comparator of a task can be assigned to any time slot. However, many applications require a bounded maximum fault latency l, where l is a positive integer. The value l is decided based on a real-time deadline, the frequency of an output, or other similar constraints. Again, our schedule should detect every fault that manifests itself.

Given $m = 2^h$ processors, a complete binary tree $G(p)$ and a bounded maximum fault latency l, we modify our fault-secure schedule to ensure that the distance between a task, its duplicate and the comparator is no more than l. Note that in order to ensure that the maximum fault latency is bounded more tests are needed than in a fault-secure schedule which does not consider fault latency. Depending on the values of l, m and p, additional time slots may or may not be required. Following Theorem 4, we modify our fault-secure scheduling based on four different cases.

• Case 1: $p \leq h$
When $l = 1$, i.e. the fault latency is one unit, each task and its duplicate should be assigned to the same time slot and they should be checked at the next time slot. If $p = h$ then there are only 2^{p-1} empty entries at slot 2 while there are 2^p tasks at slot 1. Therefore, one additional time slot needs to be added between slots 1 and 2. In general, there are $2^{p-(i-1)}$ tasks at slot i and $2^p - 2^{p-i}$ empty entries at slot $i + 1$. Since $2^{p-(i-1)} < 2^p - 2^{p-i}$ (for $i > 1$), there is no need for additional time slots to test tasks in slots other than slot 1. There are enough empty entries where tests can be assigned. If $p < h$, then there are $2^{h-1} \geq 2^p$ empty entries in slot 2, i.e. no additional time slot is needed. When $1 < l < p$, comparisons are needed every l time slots. An efficient way is to check only those tasks scheduled at time slot kl, $k = 1, 2, \ldots, \lfloor p/l \rfloor$. Clearly, there are more empty entries at slot $kl + 1$ than the number of tasks at slot kl. Hence, there is no need for additional time slots to schedule additional tests in this case. Finally, when $p \leq l$ nothing needs to be changed, because the depth of the computation tree is shorter than the given maximum fault latency.

• Case 2: $1 = h < p$
In this case only two processors are used and there are no empty entries. Any additional tests require additional time slots. Again, it suffices to check tasks at slots kl, $k = 1, 2, \ldots, \lfloor p/l \rfloor$. This results in a total of additional time slots $\lfloor p/l \rfloor$.

• Case 3: $1 < h < p \wedge h + \log(h-1) \geq p$
This case is shown in Figure 7, where the number of idle entries is more than or equal to the number of duplicates of all the tasks from G_{L_i}, where $1 \leq i \leq m$ and $m = 2^h$ tests. Let us first calculate the maximum fault latency. We consider two types of nodes: nodes from G_T and nodes from G_{L_i}. The maximum fault latency among nodes from G_T is h (see Figure 7) and among nodes from G_{L_i} is $2^{p-h} - 1 + s$, where s is the number of time slots that have idle entries used to fill the $2^h(2^i - 1)$ duplicates of nodes from G_{L_i} and the 2^h tests. Based on Figure 7, s satisfies the following equation:

$$\begin{aligned} 2^h(2^i - 1) + 2^h = 2^p \\ &= (2^h - 2^h) + (2^h - 2^{n-1}) \\ &\quad + (2^h - 2^{n-2}) + \ldots + (2^h - 2^{n-(s-1)}) \end{aligned}$$

then

$$2^{p-h} + 1 < s \leq 2^{p-h} + 2.$$

Therefore, the maximum fault latency among nodes from G_{L_i} is $2^{p-h+1} + 1$. Overall, the maximum fault latency is:

$$\max\{2^{p-h+1} + 1, h\}.$$

Based on the value of l, the given maximum fault latency, we consider the following subcases:

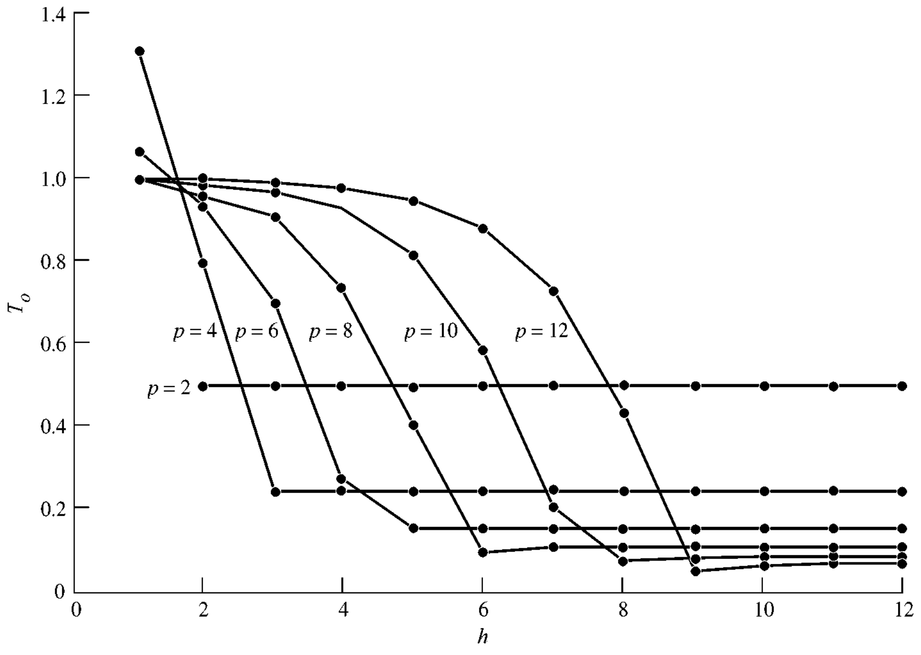


FIGURE 10. Curves of T_o for some fixed values of p.

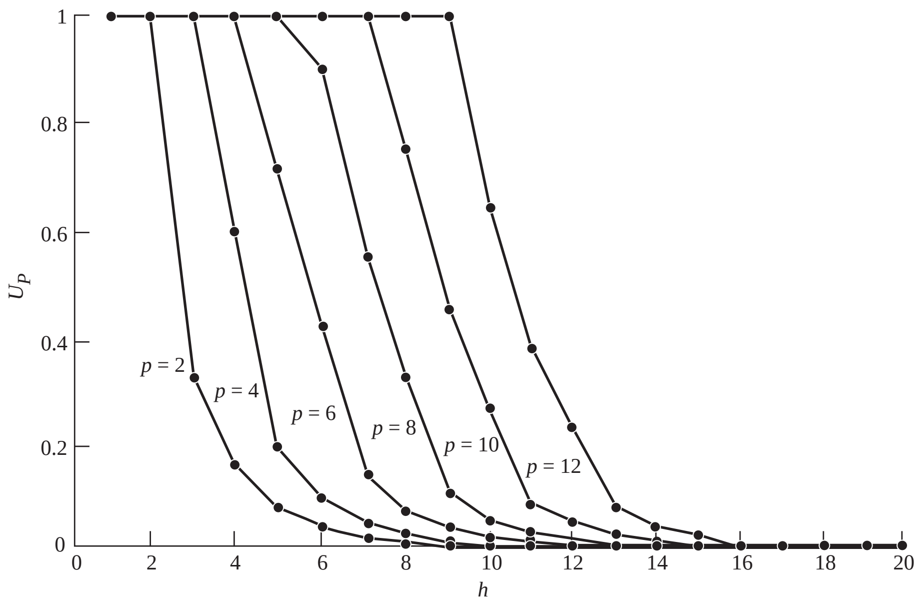


FIGURE 11. Curves of U_P for some fixed values of p.

– subcase 1: $l \geq \max\{2^{p-h+1} + 1, h\}$

In this case, nothing needs to be done.

– subcase 2: $2^{p-h+1} + 1 \geq l < h$

In this case, the given maximum fault latency is greater than or equal to the maximum fault latency among nodes from G_{L_i} but smaller than the maximum fault latency among nodes from G_T. It is easy to see that there are enough idle entries in the original fault secure schedule to accommodate additional tests required for nodes in G_T (again we only need to conduct tests at every l time slots), and the original assignment remains unchanged.

– subcase 3: $l < 2^{p-h+1} + 1$

This corresponds to the case where the fault latency among nodes in G_{L_i} is longer than the given maximum fault latency. That is, the shaded area can no longer be used to assign duplicates and tests of G_{L_i}. Additional time slots have to be inserted early in the schedule to ensure that the original task, its duplicate and the tests are within l distances. The worst case happens when $2^{p-h} - 1$ additional time slots are used for duplicates and $\lfloor (2^{p-h} - 1)/l \rfloor$ additional time slots are tests. For nodes from G_T, it suffices to add additional $\lfloor h/l \rfloor$ tests and there are enough idle entries to accommodate them (again we only need to test nodes at every l slots); therefore, no additional time slots are required for these nodes.

• Case 4: $1 < h < p \wedge h + \log(h-1) \geq p$
This can be treated as similar to case 3, except that because $h + \log(h-1) \geq p$, the maximum fault latency is

$$\max\{2^{p-h+1}+1, h\} = 2^{p-h+1}+1.$$

Subcase 2 of case 3 will never occur.

6.2. Non-unit length tasks and computation graphs of arbitrary binary trees

So far we have only considered application graphs which are complete binary trees (each node has zero or two children) with unit length tasks. To relax this condition, we consider cases for arbitrary binary trees (where each node has zero, one or two children) with unit length tasks. Any arbitrary binary tree with non-unit length tasks can be converted into another arbitrary binary tree with unit length tasks. This can be easily done by replacing each non-unit node of length l with a sequence of l unit nodes.

The concept of the level of nodes is also extended. Instead of assigning each node a level, we assign a range of levels to each node: $[L_{\min}, L_{\max}]$, where $L_{\min}$ is the distance between this node and its farthest leaves and $L_{\max}$ is the depth of the tree less the distance between the root and this node. A node can be assigned to any level within the range $[L_{\min}, L_{\max}]$. Clearly, without the restriction on the number of processors used, Theorem 1 still applies. That is, the minimum total execution time of a fault-secure schedule for $G(p)$ is still p, where p is the length of the longest path from the root to one of the leaves. Because arbitrary binary trees are generally 'thinner' than the corresponding complete binary tree of the same depth, we might use fewer processors to ensure a fault-secure scheduling without increasing the execution time. To generate a 'thinnest' schedule, we should assign nodes to levels such that the maximum number of nodes in levels is minimized; however, it is an NP-complete problem to find the 'thinnest' schedule. More formally, each node is assigned to a level within its range such that the following expression is minimized:

$$\max_{1 \leq i \leq p} \{|L_i|\}.$$

Suppose the value obtained from the above expression is k, which is smaller than 2^p. Then we only need $2k$ processors and to apply the basic fault-secure algorithm to ensure fault security while its execution is bounded by $p+1$. Due to the irregularity of the structure of the arbitrary binary tree, there is no systematic way of scheduling to ensure optimality, rather we provide here only the general guideline.

In the example of Figure 4, both nodes N_{22} and N_{24} can be assigned to L_2 or L_3. The thinnest schedule occurs only when these nodes are assigned to L_2 (see Figure 4b). Note that one potential problem with this approach occurs when it is impossible to logically split a task into many subtasks.

6.3. Multiple errors

Throughout the paper, we use the 1-fault-secure condition, based on the assumption that multiple faults rarely occur in a parallel/distributed system within a (relatively short) time period. If the time period is sufficiently short, multiple faults are reduced to simultaneous faults (faults that occur at the same time) only. The validity of the 1-fault-secure condition depends very much on how well we can control the length of the period. More specifically, fault latency, which is measured by the time between the occurrence of a fault and the detection of this fault, should be reduced. We assume that once a fault is detected, the faulty processor will be soon detected and replaced so that a fault in one period will not be propagated to the next period to cause multiple faults.

If all the above approaches fail, we have to consider fault-secure schedules that can cover multiple faults at the expense of using additional processors and time slots. In general, to ensure k-fault security at least $(k+1)$-version computation is required. Under the assumption that each processor that tests versions can fail, we need at least k tests. To show the fault security of a schedule, we need to prove one of the following two conditions.

- At least one test generates a correct decision: pass when all the versions are correct or fail when at least one version is incorrect.
- All the versions are correct when all tests generate incorrect decisions.

Based on the above two conditions, we have the following way to produce a general k-fault-secure schedule. When k tests are used, they are assigned to k different processors and they do not share processors with any of the $k+1$ versions to be tested. Therefore, a total of $2k+1$ processors is used in this case. When $k+1$ tests are used, they can share the $k+1$ processors with the $k+1$ versions. In this case, only $k+1$ processors are used. Notice that when $k > 1$, the test used is a majority voting and now the output comes directly from the voted result. Since majority voting also masks faults, the system obtained not only assures k-fault security but also achieves fault tolerance.

We study the case when $k = 2$ to illustrate our extended fault-secure scheduling. Let us assume the three versions are labeled N_1, N_2 and N_3 and tests are labeled t_1, t_2 and t_3 (if they are required). We start with the schedule of tests. Based on the above discussion, tests could either be assigned to a fourth and a fifth processor (two tests are required as shown in Figure 12a) or share processors with three versions (three tests are required as shown in Figure 12b). We first show that the two configurations in Figure 12 are both 2-fault secure and then we show that the number of tests used in both cases is minimum.

In Figure 12a, if one of the two processors (P_4 and P_5) on which the two tests are assigned is healthy, any faults from the three versions (N_1 on P_1, N_2 on P_2 and N_3 on P_3) will be caught or masked by the correct test; otherwise, both processors (P_4 and P_5) are faulty and both tests could

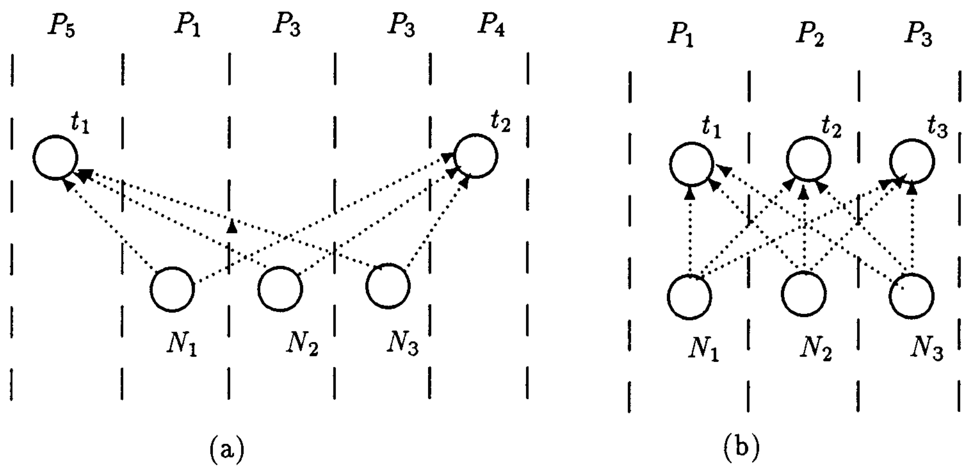


FIGURE 12. The assignment of tests (a) to a fourth and a fifth processor and (b) to processors used.

generate any decision (pass or fail). Because all three versions (on the healthy processors) produce correct results, the final outcome is still safe. Assume now that we only have one test which is assigned to a fourth processor P_4, and that P_4 and one of the processors on which one of the three versions is assigned, say P_1, are faulty. Since the faulty test could generate a pass decision, the outcome is not safe when version N_1 is incorrect. Therefore, at least two tests are required in this configuration.

In Figure 12b where the three tests and the versions share three processors, at least one test always generates a correct decision: pass when all three versions are correct or fail when at least one version is incorrect. Because the system terminates execution whenever a test generates a fail decision, this configuration is safe. Assume we only have two tests and they share the same processors with the three versions. Without loss of generality, assume that t_1 and N_1 share processor P_1, t_2 and N_2 share processor P_2 and N_3 is assigned to N_3. Suppose that both P_1 and P_2 are faulty; both t_1 and t_2 could generate pass decisions while N_1 and N_2 could be incorrect. Therefore, at least three tests are required in this configuration.

Once the basic configurations are identified, we can schedule the given computation graph based on the approach proposed in the previous sections. The details are beyond the scope of this paper.

When k increases, the number of tests and versions also increases, which makes our approach too costly for large k. We can use one of the three following approaches to alleviate this problem.

• Decrease k by shortening the period
Here we assume that there are at most k faults in the system within a specified time period. That is, this period is the maximum length of time used to complete all the versions of a task and their corresponding tests. If we can assign all the versions of a task together with their tests within a short period of time, we decrease the value of k.

• Weaken the fault model
In our model we use a relatively strong fault model by assuming that a fault could only be either permanent or transient. By weakening the fault model, we can reduce the number of versions or tests required. For example, by assuming that all faults are independent (or equivalently, that there are no correlated faults), two versions and one correct comparator are enough to ensure k-fault security for any k.

• Combine with other approaches
Algorithm-based fault tolerance (ABFT), [30] and [31], encodes data at the system level in the form of some error-correcting or detecting code. Most errors can be detected in this scheme and in general, computations are not duplicated. To ensure absolute fault security, we can combine our approach with ABFT. ABFT reduces the value of k while our fault-secure scheduling ensures absolute fault security.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have analyzed two basic problems of fault-secure scheduling for multiprocessor systems. Given a set of computational tasks of unit length expressed as a complete binary tree, first the tasks are scheduled such that the total execution time is a minimum and no undetected single-error result will be delivered. Second, given a deadline, an algorithm is developed which generates a fault-secure schedule using a minimum number of processors. These algorithms include those in [14] and [15] as special cases.

In [32], two types of errors were considered. In the first type (called contaminating errors), the use of an erroneous operand in a given operation necessarily causes the result of the operation to be incorrect. In the second type (called redeemable errors), the use of an erroneous operand need not cause the result of an operation to be erroneous. One could extend the proposed schemes using those fault types.

Another possible direction is to apply other fault-tolerant mechanisms to ensure fault-secure computations and to increase system availability. The current approach does not provide reconfiguration capabilities, i.e. the system stops whenever a failure is detected. In the roll-forward recovery approach, [18] and [19], a failure is detected by a comparison mismatch that triggers a validation step. However, the processors continue execution and a spare (third processor) is used to determine which of the divergent processors is correct. In this way, the availability of the system is improved.

A more interesting study would be an investigation of trade-offs in schedule length, number of processors and number of faults in the system. To tolerate more faults (within a given period), we can partition the period into many small periods so that each period, with a high probability, contains at most one fault. Based on the proposed algorithms, the fault latency is upper bounded by the depth of the given tree. One approach is to partition the given tree into many small trees, where each of them meets the given constraint on its depth (i.e. fault latency). The cost of this approach is that the overall schedule length may increase when we put together all these trees as one schedule. To investigate different trade-offs, we should probably start with some sample examples in existing computer aided scheduling packages, such as CASCH [33] and PYRROS [34]. Once some useful insights are obtained, we may verify them through simulation (rather than mathematical analysis since close form solutions are unlikely).

ACKNOWLEDGEMENTS

We are grateful to the anonymous referees for their helpful comments and suggestions which improved the quality of this paper.

REFERENCES

[1] Lipovski, G. J. and Malek, M. (1987) *Parallel Computing: Theory and Comparisons*, Wiley, New York.

[2] Siegel, H. J. *et al*. (1992) Report of the purdue workshop on grand challenges in computer architecture for the support of high performance computing. *J. Parallel Distrib. Comput.*, **13**, 199–211.

[3] Bokhari, S. H. (1981) On the mapping problem. *IEEE Trans. Comput.*, **30**, 207–214.

[4] Chaudhary, V. and Aggarwal, J. K. (1993) A generalized scheme for mapping parallel algorithms. *IEEE Trans. Parallel Distrib. Syst.*, **4**, 328–346.

[5] Chu, W. W., Holloway, L. J., Lan, M. T. and Efe, K. (1980) Task allocation in distributed data processing. *Computer*, **13**, 57–69.

[6] Fernandez, E. B. and Bussell, B. (1973) Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trans. Comput.*, **22**, 745–751.

[7] Kruatrachue, B. and Lewis, T. (1988) Grain size determination for parallel processing. *IEEE Software*, **5**, 23–31.

[8] Lo, V. M. (1988) Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, **37**, 1384–1397.

[9] Gonzalez, Jr, M. J. (1977) Deterministic processor scheduling. *ACM Computing Surveys*, **9**, 173–204.

[10] Ramamoorthy, C. V., Chandy, K. M. and Gonzalez, M. J. (1972) Optimal scheduling strategies in a multiprocessor system. *IEEE Trans. Comput.*, **21**, 137–146.

[11] Kuhl, J. G. and Reddy, S. M. (1986) Fault-tolerance considerations in large, multiple-processor systems. *Computer*, **19**, 56–67.

[12] Geist, R. and Trivedi, K. (1990) Reliability estimation of fault-tolerant systems: tools and techniques. *Computer*, **23**, 52–61.

[13] Laprie, J. C. (1989) Dependability: a unifying concept for reliable computing and fault tolerance. In Anderson, T. (eds) *Dependability of Resilient Computers*, pp. 1–28. BSP Professional Books, London.

[14] Banerjee, P. and Abraham, J. A. (1984) Fault-secure algorithms for multiple processor systems. *Proc. 11th Int. Symp. on Computer Architecture*, Ann Arbor, MI, June, pp. 279–287. ACM Press, New York.

[15] Gu, D., Rosenkrantz, D. J. and Ravi, S. S. (1991) Construction and analysis of fault-secure multiprocessor schedules. *Proc. 20th Int. Symp. on Fault Tolerant Computing Systems*, pp. 120–127.

[16] Wu, J. (1991) A fault-tolerant task scheduling method for parallel processing systems. *Int. J. Mini and Microcomputers*, **13**, 135–138.

[17] Siewiorek, D. P. and Swarz, R. S. (1992) *Reliable Computer Systems – Design and Evaluation* (2nd edn), Digital Press, Burlington, MA.

[18] Long, J., Fuchs, W. K. and Abraham, J. A. (1992) Forward recovery using checkpointing in parallel systems. *Proc. 1992 Int. Conf. Parallel Processing*, Vol. 1, pp. 272–275. CRC Press, Boca Raton, FL.

[19] Pradhan, D. K. and Vaidya, N. H. (1992) Roll-forward checkpointing scheme: concurrent retry with nondedicated spares. *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 166–174. IEEE Computer Society Press, Los Alamitos, CA.

[20] Johnson, B. W. (1989) *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, Reading, MA.

[21] Fabre, J. C., Deswarte, Y., Laprie, J. C. and Powell, D. (1988) Saturation: reduced idleness for improved fault-tolerance. *Proc. 18th Int. Symp. on Fault-Tolerant Computing*, pp. 200–205. IEEE.

[22] Krishna, C. M. and Shin, K. G. (1986) On scheduling tasks with a quick recovery from failure. *IEEE Trans. Comput.*, **35**, 448–455.

[23] Liestman, A. L. and Campbell, R. H. (1986) A fault-tolerant scheduling problem. *IEEE Trans. Software Eng.*, **12**, 1089–1095.

[24] Srinivasan, A. and Shoja, G. C. (1991) A fault-tolerant scheduler for distributed real-time systems. *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp. 219–222. IEEE Computer Society Press, Los Alamitos, CA.

[25] Ahmad, I. and Kowk, Y.-K. (1994) A new approach to scheduling parallel programs using task duplication. *Proc. 1994 Int. Conf. on Parallel Processing*, Vol. II, pp. 47–51. CRC Press, Boca Raton, FL.

[26] Darbha, S. and Agrawal, D. P. (1994) A task duplication based optimal scheduling algorithm for variable execution time tasks. *Proc. 1994 Int. Conf. on Parallel Processing*, Vol. II, pp. 52–56. CRC Press, Boca Raton, FL.

[27] Luque, E., Ripoll, A., Margalef, T. and Hernandez, P. (1993) Static scheduling of parallel program graphs including loops. *Proc. 26th Ann. Hawaii Int. Conf. on System Sciences*,

pp. 526–534. IEEE Computer Society Press, Los Alamitos, CA.

[28] Hu, T. C. (1961) Parallel sequencing and assembly line problems. *Operat. Res.*, **9**, 841–848.

[29] Dai, D., Wu, J. and Fernandez, E. B. (1992) *Optimal Fault–Secure Scheduling Algorithms for Multiprocessor Systems*. Technical Report, TR-CSE-92-17, Florida Atlantic University.

[30] Huang, K. H. and Abraham, J. A. (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, **C-33**, 518–528.

[31] Vinnakota, B. and Jha, N. K. (1993) Diagnosability and diagnosis of algorithm-based fault-tolerant systems. *IEEE Trans. Comput.*, **C-42**, 925–937.

[32] Gu, D., Rosenkrantz, D. J. and Ravi, S. S. (1992) Fault/error models and their impact on reliable multiprocessor schedules. *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 176–184. IEEE Computer Society Press, Los Alamitos, CA.

[33] Ahmad, I., Kwok, Y.-K., Wu, M.-Y. and Shu, W. (1997) Automatic parallelization and scheduling of programs on multiprocessors using CASCH. *Proc. 1997 Int. Conf. on Parallel Processing*, pp. 288–291. IEEE Computer Society Press, Los Alamitos, CA.

[34] Yang, T. and Gerasoulis, A. (1992) PYRROS: static task scheduling and code generation for message-passing multi-processors. *Proc. 6th ACM Int. Conf. on Supercomputing, 1992*. IEEE Computer Society Press, Los Alamitos, CA.