PAPER

Error Models and Fault-Secure Scheduling in Multiprocessor Systems

Koji HASHIMOTO[†], Tatsuhiro TSUCHIYA[††], *and* Tohru KIKUNO[††], *Regular Members*

SUMMARY A schedule for a parallel program is said to be 1-fault-secure if a system that uses the schedule can either produce correct output for the program or detect the presence of any faults in a single processor. Although several fault-secure scheduling algorithms have been proposed, they can all only be applied to a class of tree-structured task graphs with a uniform computation cost. Besides, they assume a stringent error model, called the redeemable error model, that considers extremely unlikely cases. In this paper, we first propose two new plausible error models which restrict the manner of error propagation. Then we present three fault-secure scheduling algorithms, one for each of the three models. Unlike previous algorithms, the proposed algorithms can deal with any task graphs with arbitrary computation and communication costs. Through experiments, we evaluate these algorithms and study the impact of the error models on the lengths of fault-secure schedules.

key words: *multiprocessors, fault-secure scheduling, task graphs, error models, tests*

1. Introduction

This paper focuses on fault-secure multiprocessor scheduling. The goal of fault-secure scheduling is to detect errors in computation of parallel programs carried out on multiprocessor systems. The basic approach to achieving fault security is to duplicate every task of a program and compare outputs of copies to ensure that either the output of the program is correct or at least one of the comparisons reports the existence of errors.

The concept of fault security was originally introduced in logic circuit design [16]. A circuit is *fault-secure* if for any single fault within the circuit, the circuit either produces correct output or produces a non-codeword. Banerjee and Abraham [4] first applied this concept to multiprocessor scheduling. Gu et al. [12] have further investigated the formal characterization of fault-secure multiprocessor schedules by introducing the concept of *k-fault-secure scheduling.* In a k-fault-secure schedule, the output of a system is guaranteed to be either correct or tagged as incorrect for up to k processor faults. In their model, a parallel program is composed of a set of tasks and represented by a directed acyclic graph, and the number of processors is unlimited. Some scheduling algorithms for achieving 1-fault-security have been proposed in [12]. More recently, Wu et al. [29] proposed an optimal fault-secure scheduling algorithm. Given the number of processors, the algorithm generates a 1-fault-secure schedule with the minimum schedule length.

In both [12] and [29], the fault-secure scheduling technique has been used to make a computation fault-secure against *redeemable errors* [13] in a system. However, different applications may require different degrees of fault security. For that reason, we introduce two more error models in addition to the redeemable error model. By restricting the manner of error propagation, these new models exclude unlikely errors that are assumed in the redeemable error model. We present three scheduling algorithms for achieving 1-fault-security, one for each error model. Of the three algorithms, the two that assume the new models are designed so as to exploit the restriction on errors to shorten schedule length. By conducting simulations, we study the effects of the error models on 1-fault-secure schedules in multiprocessor systems.

The previous algorithms proposed in [12] and [29] assume that communication costs are negligible and all tasks have a uniform unit execution time. Moreover, these algorithms can only be applied to a class of tree-structured task graphs. In contrast, we consider parallel programs represented by directed acyclic graphs with arbitrary computation and communication costs. Multiprocessor scheduling for most precedence-constrained task graphs is an NP-complete problem in its general form [9]. The algorithms we propose in this paper are heuristic; that is, schedules they produce are not necessarily optimal.

It is well known that inter-processor communication makes serious effects on the performance of parallel processing. *Task duplication* [19], [20] is an effective technique to reduce overheads of the communication and improve the performance. In this technique, a duplicated task is allocated to the same processor as one of its succeeding tasks so that it can send data to the successor without communication delay. Task duplication thus improves the start times of tasks that need to wait for their preceding tasks, and also improves the finish time of the given program consequently.

In our approach to achieving 1-fault-security, every task in a given task graph is replicated, and equal-

Manuscript received December 22, 1999.
Manuscript revised August 7, 2000.

[†]The author is with Hitachi Laboratory, Hitachi Ltd., Hitachi-shi, 319–1292 Japan.
[††]The authors are with the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560–8531 Japan.

ity tests are carried out between the copies. Our algorithms schedule copies of tasks based on the task duplication technique for achieving better performance while maintaining the 1-fault-secure property.

The remainder of this paper is organized as follows. The system model assumed in this paper is described in Sect. 2. In the section, the three error models are also described, and for each model we formally define a fault-secure schedule. In Sect. 3, 1-fault-secure scheduling algorithms are proposed under the models. The correctness proof of the algorithms is given in Sect. 4. The results of simulation studies are shown in Sect. 5. In Sect. 6, related work in the field is overviewed. The paper concludes with Sect. 7.

2. Preliminaries

2.1 System and Task Model

We consider a multiprocessor system that consists of n identical processing elements (PEs) and that runs one application program at a time. All PEs are fully connected with each other via a reliable network. A PE can execute tasks and communicate with another PE at the same time. This is typical with dedicated I/O processors and direct memory access.

A parallel program is represented by a weighted directed acyclic graph (DAG) $G = (V, E, w, c)$, where V is the set of nodes and E is the set of edges. Each node represents a task v, and is assigned a computation cost $w(v)$, which indicates the task execution time. Each edge $< v, v' > \in E$ from v to v' corresponds to the precedence constraint that task v' cannot start its execution before receiving all necessary data from task v. Given an edge $< v, v' >$, v is called an *immediate predecessor* of v', while v' is called an *immediate successor* of v. If there exists a path from v' to v, v' is called a *predecessor* of v. A task that has no immediate successors is called an *output task*. Each edge is assigned a communication cost $c(v, v')$, which indicates the time required for transferring necessary data between different PEs. If the data transfer is done within the same PE, the communication cost is zero. In the following, we call such a weighted DAG a *task graph*. Various applications are known to be represented by weighted DAGs (e.g., [24]). Figure 1 shows examples of task graphs. In the figure, the number adjacent to each node indicates the execution time of the task represented by the node, and the number on each edge is the communication cost for data transfer.

We introduce some definitions and terminology as in [15]. For a path in a task graph, its *length* is defined as the summation of task execution times along the path excluding communication delays. The *level* of a task is defined as the length of the longest path from the node representing the task to a node that has no successor nodes. In Fig. 1 (a), for example, the levels of

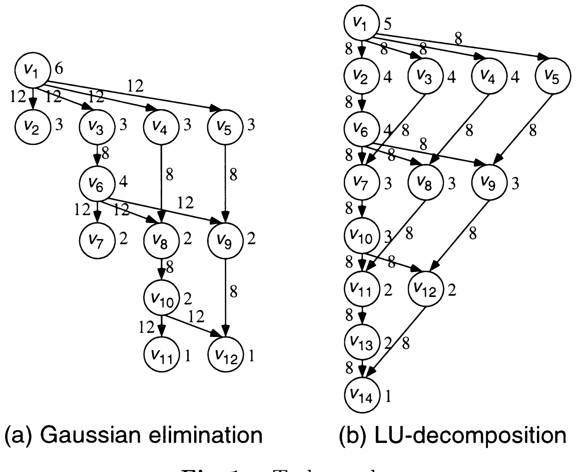


Fig. 1 Task graphs.

v_6 and v_7 are 9 and 2, respectively. Finally, the *height* of a task is defined as

$$height(v) = \begin{cases} 0, & U = \emptyset, \\ 1 + \max_{u \in U}\{height(u)\}, & U \neq \emptyset, \end{cases}$$

where U is a set of immediate successors of v. In Fig. 1 (a), for example, the heights of v_6 and v_7 are 3 and 0, respectively.

2.2 Scheduling

In general, multiprocessor scheduling refers to the process in which tasks in a task graph are assigned to PEs and the time slots in which the tasks are executed are determined. When more than one copy is allowed to be scheduled for each task, it is also necessary to specify from which copies to which copies data are transferred.

Since as discussed below, not only copies of tasks but also tests may be scheduled for the purpose of fault security, a schedule S we consider here is a tuple (σ, ϕ) where σ is a set of scheduled copies of tasks and tests, and ϕ is a mapping function that determines for a given element of σ its corresponding task or test, assigned PE, time slot, and the subset of σ from which the element receives data.

To distinguish between a task $v \in V$ and its copies scheduled actually, we call the latter the *instances* of v. We represent by $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ the fact that the instance s of v receives necessary data from $\alpha_1, \alpha_2, \ldots, \alpha_r$, which are instances of the immediate predecessors of v. By definition, r is equal to the number of immediate predecessors. We always write α_i's in ascending order of the indices of their corresponding tasks.

Fault-secure scheduling, as discussed here, refers to producing a schedule such that even if any single fault occurs, the system can produce the correct result or detect the fault. We call such a schedule a *1-fault-secure schedule* [12]. The goal of our research is to minimize the schedule length while achieving 1-fault-security.

In our approach to achieving the 1-fault-secure property, every task $v \in V$ is replicated to produce at least two copies of its output and equality tests are carried out between different copies of tasks. To do so, we need to allocate *tests* to PEs, in addition to the normal tasks of V. A test reports either "*equal*" or "*not equal*" according to the equality of the outputs of the copies compared. A fault is detected when a test reports "*not equal*". We assume that the outcome of a test carried out on a fault-free PE is always correct.

We use the notation $\tau(\alpha_1, \alpha_2, \ldots, \alpha_m)$ to indicate a test that compares the outputs of instances $\alpha_1, \alpha_2, \ldots, \alpha_m$ of the same task. Each test requires time for execution and receiving data for comparison also incurs a communication delay. In this paper, we do not introduce notations to represent these costs, in order to prevent them from deteriorating readability. In the rest of the paper, we will carefully proceed without them, while keeping the correctness of discussions.

2.3 Error Models and 1-Fault-Secure Schedules

We assume that a fault in a PE can result in errors in the outputs of an arbitrary set of instances of tasks and tests allocated to the PE. We call such a set a *fault pattern* if it is not empty; that is, a fault pattern is a non-empty subset of instances of tasks and tests that consists of all instances whose outputs can be made erroneous directly by a fault in the system.

If a task receives erroneous outputs of other tasks, then the task itself may or may not become erroneous. That is, an error in an instance of a task causes a (possibly empty) subset of instances that receive data from that instance to become erroneous. The three error models we consider are different in their assumptions regarding the values that can be taken by erroneous output. The first error model is the most general, and is equivalent to the *redeemable error* model in [13]. This model does not impose any assumption on the output values of tasks that have received erroneous values. By adding an assumption that restricts the output values of such tasks, we define another error model called the *Type A error* model. By imposing another assumption, we define more benign errors. We refer to this error model as the *Type B error* model.

In the following, we describe these error models precisely, and then give a formal definition of 1-fault-secure schedules under the error models. First, as in [12], [13], we introduce the notion of *interpretation*, which represents a possible scenario.

Definition 1 (Interpretation): Given a schedule S, an *interpretation* I for S is a set $\Sigma = \{c, e, n, \mu_1, \mu_2, \ldots\}$ of *labels*, with distinguished labels "c", "e", "n", together with an assignment of a label to each instance in S such that:

1. each instance of a task is assigned a label from $\Sigma - \{e, n\}$, and
2. each instance of a test is labeled either "e" or "n."

In the definition, "c" means "*correct*," whereas μ_i represents an erroneous value of an output. Therefore, an instance of a task labeled "c" produces a correct output value, while an instance assigned a label μ_i outputs an erroneous value. The labels "e" and "n" represent the two possible outcomes of a test, "*equal*" and "*not equal*" respectively. In the following, we use $Label_I(s)$ as the label assigned by an interpretation I to an instance s in S.

Under different error models, a given fault pattern may induce different scenarios. The following definition specifies the three error models, and gives the rule for producing scenarios for a given fault pattern under a given error model.

Definition 2 (Consistency of Interpretation): Suppose that a schedule S and a fault pattern P' are given.

[Redeemable error] An interpretation I of S is *consistent* with P' under the redeemable error model if and only if the following conditions are satisfied:

(A) for an instance s of a task, if $Label_I(s) \neq$ "c," then either $s \in P'$ or there is at least one instance α in $D(s)$ such that $Label_I(\alpha) \neq$ "c."

(B) for an instance t of a test $\tau(\alpha_1, \alpha_2, \ldots, \alpha_m)$, if $Label_I(t) =$ "e," then either $t \in P'$ or $Label_I(\alpha_1) = Label_I(\alpha_2) = \cdots = Label_I(\alpha_m)$.

(C) for an instance t of a test $\tau(\alpha_1, \alpha_2, \ldots, \alpha_m)$, if $Label_I(t) =$ "n," then either $t \in P'$ or $Label_I(\alpha_i) \neq Label_I(\alpha_j)$ for some α_i and α_j ($i \neq j, 1 \leq i, j \leq m$).

(D) for two instances, s and s', of a task, with $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \ldots, \alpha'_r)$, if $s, s' \notin P'$ and $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q ($1 \leq q \leq r$), then $Label_I(s) = Label_I(s')$.

[Type A error] An interpretation I of S is *consistent* with P' under the Type A error model if and only if conditions (A), (B), (C), (D), and the following condition are satisfied:

(E) for two instances, s and s', of a task, with $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \ldots, \alpha'_r)$, if $s, s' \notin P'$ and there exists at least one α_q ($1 \leq q \leq r$) such that $Label_I(\alpha_q) \neq Label_I(\alpha'_q)$, then $Label_I(s) \neq Label_I(s')$.

[Type B error] An interpretation I of S is *consistent* with P' under the Type B error model if and only if conditions (A), (B), (C), (D), (E), and the following condition are satisfied:

(F) for two instances, s and s', of a task, if $s \notin P'$, $s' \in P'$ and $Label_I(s) \neq$ "c", then $Label_I(s) \neq Label_I(s')$.

Condition (A) implies that in any scenario for a fault pattern P', the output of s can be erroneous only if either s is computed on the faulty PE or one of the instances of the immediate predecessors of v is erroneous. Conditions (B) and (C) indicate that the outcome of a test carried out on a non-faulty PE is determined by the labels of instances participating in the test, and that a valid scenario may assign "*e*" or "*n*" arbitrarily to tests carried out on the faulty PE. Condition (D) states that different instances of a task computed on non-faulty PEs with identical input values must have the same output value. Condition (E) is the assumption corresponding to the Type A error model, which means that instances of a task computed on non-faulty PEs with different input values must have different output values. Finally, Condition (F) corresponds to the Type B error model. For two instances of a task, say s and t, the erroneous output value of s that is caused by the erroneous inputs of s is different from the erroneous output value of t that is caused directly by a fault in the PE assigned to t.

As stated before, the redeemable error model does not impose any restrictions on erroneous values. For example, consider two instances of a task that multiplies n input values. Suppose that one of the instances receives an erroneous zero value as the ith input, and the other instance receives an erroneous zero as the jth ($i \neq j$) input. In this case, the outputs of the two instances are both zero which may be an erroneous value. This case is modeled by the redeemable error model. However, for numerical computing in various fields such as fluid mechanics, chemistry, physics, design, and signal processing, each task handles a large volume of numerical data. In such a situation, it is extremely unlikely that two instances of a task output exactly the same erroneous result when they receive different inputs. It is also unlikely that an erroneous output value caused directly by a processor fault agrees with the erroneous output value of another instance that is caused by erroneous inputs. Type A and Type B error models exclude these unlikely situations from the redeemable error model.

Finally, based on the concept of interpretation, we formally define a 1-fault-secure schedule as follows.

Definition 3 (1-Fault-Secure schedule): A schedule S is 1-fault-secure under an error model if and only if for every fault pattern $P'(\neq \emptyset)$ and for every interpretation I that is consistent with P' under the error model, $Label_I(s)$ = "*c*" for every instance s of every output task, or there exists at least one instance t of a test such that $Label_I(t)$ = "*n*."

3. 1-Fault-Secure Scheduling Algorithms

In this section, we present three scheduling algorithms to achieve the 1-fault-secure properties under the error models. For the redeemable error model, we develop an original algorithm STR, since previously proposed algorithms can only handle tree-structured task graphs. Algorithms $FSAE$ and $FSBE$ are proposed for the Type A error model and the Type B error model, respectively. STR is described in Sect. 3.1, while $FSAE$ and $FSBE$ are both described in Sect. 3.2.

These algorithms tag each instance with either "0" or "1," which we call the *version number*, such that every task has its instances with different numbers. The algorithms schedule tasks in such a way that each task communicates almost exclusively with tasks with the same version number, in order to avoid contaminating data and to allow a fault to be detected.

3.1 Algorithm for the Redeemable Error Model

If Condition (E) in Definition 2 is not met, instances with different version numbers cannot be allocated to the same PE. This is because, given a fault pattern P' and an interpretation I consistent with P', two instances of an output task can have the same label not equal to "*c*," which means that the error cannot be detected by comparison. In order to achieve the 1-fault-secure property under the redeemable error model, it is therefore essentially necessary to simply duplicate a non-fault-secure schedule.

We refer to this straightforward algorithm as STR. Algorithm STR produces a fault-secure schedule by applying DSH [19], [20] twice. DSH (Duplication Scheduling Heuristic) is a normal (i.e., non-fault-secure) scheduling algorithm proposed by Kruatrachue in [19], [20]. It adopts a conventional heuristic, called *list scheduling*, which is a two-step approach. That is, in the first step, all tasks are prioritized, and in the second step, tasks are scheduled one by one according to the priority. DHS assigns priorities to tasks based on their level, and then it schedules each task by using a procedure called TDP, which will be explained later. Although we uses DSH as a subroutine in STR, it should be noted that STR would work if DSH were replaced with any other scheduling algorithm.

Below STR is described. Function $TST(TQ, S)$, which will be described later, schedules tests that check the tasks in queue TQ, after the schedule S of tasks has been determined. TQ, which will be referred to as a *test queue*, keeps tasks in descending order of their priority.

```
Algorithm $STR$
Input:   $G$, a task graph;
         $P$, a set of PEs $\{p_1, p_2, \ldots, p_n\}$ $(n \geq 2)$
Output: $S$, a 1-fault-secure schedule
Begin
  Generate a partial schedule $S_0$ for $p_1, p_2, \ldots, p_{\lfloor \frac{n}{2} \rfloor}$
  by applying Algorithm $DSH$.
  Tag every instance in $S_0$ with "0".
  Generate a partial schedule $S_1$ for $p_{\lfloor \frac{n}{2} \rfloor+1}$, $p_{\lfloor \frac{n}{2} \rfloor+2}$,
  $\ldots, p_n$ by copying instances from $S_0$.
  Tag every instance in $S_1$ with "1".
  Put all output tasks into $TQ$.
  $S := TST(TQ, S_1 \cup S_2)$
  /* Schedule tests for tasks in $TQ$ */
End
```

In a schedule generated by STR, every instance tagged with "0" exchanges necessary data only with other instances within S_0, while every instance tagged with "1" does so with other instances within S_1. In other words, each instance tagged with "0" never receives any data from instances tagged with "1," and vice versa. Clearly, therefore, the system needs to compare only the results of the output tasks. The method of scheduling tests is explained in the following subsection because it is common to Algorithms $FSAE$ and $FSBE$. In Sect. 4.1, we prove that the schedules STR generates are 1-fault-secure under the redeemable error model. Figure 2 shows an example of applying STR. In the figure, v^i denotes an instance of v tagged with "i," and $t(v)$ denotes a test in which all instances of v participate.

The time complexity of DSH is known to be $O(|V|^4)$ [19], where $|V|$ denotes the number of tasks in the task graph. As explained in Sect. 3.3, the complexity of scheduling one test is $O(|V|^2)$. Therefore, the complexity of STR is $O(|V|^4)$.

3.2 Algorithms for Type A and Type B Error Models

Condition (E) in Definition 2 is met in both Type A error model and Type B error model. Condition (E) allows instances with different version numbers to be allocated to one PE. In addition to output tasks, however, it may be necessary to test other tasks. In this section, we present two fault-secure scheduling algorithms, $FSAE$ and $FSBE$, which correspond to Type A error model and Type B error model, respectively. These algorithms examine whether a test is needed or not when each instance of a task is scheduled. Tests are scheduled after all the instances of tasks have been scheduled. All procedures except for that which determines the necessity of tests are common to both algorithms.

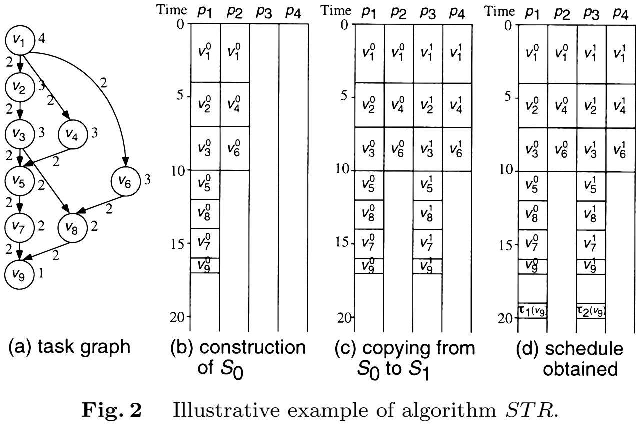


Fig. 2 Illustrative example of algorithm STR.

The algorithms schedule each task based on *task duplication* [19], [20], which can improve performance. The concept of task duplication is explained in detail in Sect. 3.4. The algorithms also tag every instance with either "0" or "1," and allocate tests by using this information. The common outline of the algorithms is as follows.

Algorithm $FSAE(FSBE)$

```
Input:   $G$, a task graph;
         $P$, a set of PEs $\{p_1, p_2, \ldots, p_n\}$ $(n \geq 2)$
Output: $S$, a 1-fault-secure schedule
Begin
  $S :=$ empty; $TQ :=$ empty
  Partitioning:
    Partition the set of tasks in $G$ into task groups
    $G_1, G_2, \ldots, G_m$ according to height.
    /*Task groups are arranged in descending order
     of height.*/
  Apply Basic algorithm BA to each task group:
    For $i = 1$ to $m$ do
      $S := BA(G_i, TQ, S)$
    End_For
    Put all output tasks into $TQ$
  Schedule tests for tasks in TQ:
    $S := TST(TQ, S)$
End
```

3.2.1 Partitioning

In both algorithms, a given set of tasks is first partitioned into subsets according to their heights in such a way that all tasks with the same height will belong to one subset. We call each subset a *task group*. For example, consider the task graph in Fig. 1 (a). The set of all tasks is partitioned into five task groups as follows.

G_1:	v_1	G_2:	v_3	G_3:	v_3, v_6
G_4:	v_5, v_8	G_5:	v_9, v_{10}	G_6:	v_2, v_7, v_{11}, v_{12}

3.2.2 Basic Algorithm

Once the program has been partitioned into task groups, the Basic algorithm described in this section is applied to each task group. This algorithm consists of two steps.

In Step 1, all tasks in a given task group are scheduled and tagged with "0." The tasks are scheduled one by one according to their priorities (the task with the highest priority is scheduled first). Priorities are assigned in descending order of level. Tasks at the same level are prioritized according to the number of their immediate successors (the task with the greatest number of immediate successors is given the highest priority).

Now suppose that $v \in G_i$ is the task to be scheduled. Note that all tasks in $G_1, G_2, \ldots, G_{i-1}$ have been already scheduled, i.e., a partial schedule S' already exists. In Step 1, v is scheduled to one of the n PEs by adding its instance, say s, to S'. All instances scheduled in Step 1, including s, are tagged with "0." The PE onto which s will be placed is determined by repeating the following process for every PE.

First, the earliest start time of s is computed, given that s runs on the PE. This can be done by calling Procedure TDP [19]. Allowing predecessor tasks to be duplicated, TDP (Task Duplication Procedure) determines the earliest time when a given task can start its execution on a designated processor. TDP also determines which tasks need to be duplicated (For details, see the appendix).

Once the start time of s on that PE has been obtained, instances from which s receives data are determined. Since every task in G_j with $j < i$ is replicated, for each immediate predecessor ip of v, there often exists more than one instance of ip that can send data to s so that s can receive the data before the earliest start time determined by TDP. For each ip, the algorithm checks whether or not an instance of ip exists that is tagged with "0" and can send data to s before the start time. If there is such an instance, it is chosen; otherwise, another instance of ip which is tagged with "1" is chosen.

If predecessors of v are duplicated by TDP in the process of scheduling s, instances which provide data to those predecessor instances are also determined as described above.

Scheduling the new instance may necessitate testing some other tasks. Based on the state of data exchanges between instances, the algorithm determines which tasks, if any, need to be tested. Such tasks are put into test queue TQ. The details of how these tasks are determined are shown for each algorithm in the next subsection.

After repeating this process for all PEs, the task is scheduled to the PE that can execute it earliest among all the PEs. If there is more than one such PE, then a PE is chosen such that the number of tasks that need to be tested is minimized.

In Step 2, all tasks in the task group are duplicated and tagged with "1". The newly duplicated copies are scheduled in the same order as in Step 1. The PE to which each is scheduled is determined in the same way as in Step 1, except that (1) an instance is never scheduled to the same PE where its corresponding task was scheduled in Step 1, and (2) instances tagged with "1" rather than "0" are chosen first as instances for receiving data. Consequently, every task is allocated to at least two different PEs.

The pseudo-code of the Basic algorithm is given below.

```
Basic algorithm $BA(G_i, TQ, S')$
Input:  $G_i$, a task group;
        $TQ$, a test queue;
        $S'$, a partial schedule
Output: $S$, a partial schedule
Begin
  Arrange tasks in $G_i$ according to their priorities
  Step 1:
    For each task $v$ in $G_i$ do
      For each PE $p$ in $P$ do
        $DTlst[p]$ := NULL /*$DTlst$ is a list
        containing duplicated predecessors of $v$.*/
        $TTlst[p]$ := NULL /*$TTlst$ is a list
        containing tasks that need to be tested.*/
        $ST[p] := TDP(v, p, DTlst[p])$ /*$ST[p]$ is
        the earliest start time of $v$ on $p$.*/
        $TTlst[p] := CKT(v, p, DTlst[p], S')$ /*Find
        tasks that need to be tested, and put them
        into $TTlst[p]$.*/
      End_For
      $p_t$ := the PE whose $ST[p_t]$ is the smallest
      Schedule $v^0$ with $DTlst[p_t]$ to $p_t$ at time $ST[p_t]$
      Put tasks in $TTlst[p_t]$ into $TQ$.
    End_For
  Step 2:
    For each task $v$ in $G_i$ do
      $p_a$ := the PE to which $v$ has been scheduled
      in Step 1
      For each PE $p$ in $P - \{p_a\}$ do
        $DTlst[p]$ := NULL; $TTlst[p]$ := NULL
        $ST[p] := TDP(v, p, DTlst[p])$
        $TTlst[p] := CKT(v, p, DTlst[p], S')$
      End_For
      $p_t$ := the PE where $ST[p_t]$ is the smallest
      Schedule $v^1$ with $DTlst[p_t]$ to $p_t$ at time $ST[p_t]$
      Put tasks in $TTlst[p_t]$ into $TQ$.
    End_For
End
```

3.2.3 Necessity of Tests

Algorithms $FSAE$ and $FSBE$ are different only in the method used to check the necessity of tests.

For each algorithm, the tasks to be tested are determined as follows. Suppose that an instance s of a task v is scheduled on a PE p. Let i be the version number of s ($i = 0, 1$).

- $FSAE$ tests every predecessor α of v that satisfies one of the following two conditions:
  - **(i)** α is an immediate predecessor of v and s receives data from an instance of α that is tagged with "$1 - i$", or
  - **(ii)** α is a predecessor of v and an instance of α that is tagged with "$1-i$" is already assigned to p.
- $FSBE$ tests every immediate predecessor α of v that satisfies only the condition (i) described above.

Under the Type B error model, even if tasks exist that satisfy Condition (ii), no tasks need to be tested because of Condition (F) in Definition 2. Therefore, one can easily expect that in most cases, $FSAE$ requires the testing of more tasks than does $FSBE$.

Because of the similarity between these two methods, we show below only the pseudo-code for the checking algorithm used in $FSAE$.

Algorithm for checking the necessity of tests $CKT(v_a, p_a, DTlst[p_a], S')$

```
Input:  $v_a$, an assigned task;
        $p_a$, a candidate PE for assignment;
        $DTlst[p_a]$, a list of duplicated tasks;
        $S'$, a partial schedule
Output: $TTlst[p_a]$, a list of tasks that need to be tested
Begin
  For each instance $v$ in $DTlst[p_a] \cup \{v_a\}$ do
    Check whether Condition (i) holds or not:
      For each immediate predecessor $\alpha$ of $v$ do
        flag := NECESSARY
        If $\alpha$ is in $DTlst[p_a]$ Then
          flag := UNNECESSARY
        Else
          For each instance $\alpha_i$ of $\alpha$ in $S'$ do
            If (the arrival time of data from $\alpha_i$ to $v$
                $\leq$ the start time of $v$ on $p_a$) and
               (the version number of $\alpha_i$ = that of $v$)
            Then
              flag := UNNECESSARY; Break
            End_If
          End_For
        End_If
        If (flag = NECESSARY) Then put $v$ into
        $TTlst[p_a]$
      End_For
    Check whether Condition (ii) holds or not:
      For each predecessor $x$ of $v$ do
        If ($x$ is assigned to $p_a$) and
        (the version number of $x \neq$ that of $v$)
          Then
            put $x$ into $TTlst[p_a]$
          End_IF
      End_For
  End_For
End
```

3.2.4 Scheduling of Tests

After the instances of all tasks have been scheduled, tests for the tasks in TQ are scheduled. A test for a task v is assigned to a PE p such that neither instances of v nor instances of its predecessors are assigned to p. If there is more than one qualifying PE, the one on which the test can be executed earliest is selected.

If there is no such PE, the test is duplicated and scheduled in such a way that each of the two copies is executed on a different PE.

All the instances of v in S participate in the test. Note that there may be more than two instances of v in S, because v may be duplicated by Procedure TDP as successors of v are scheduled. Therefore, tests are not necessarily binary equality checks, unlike in [12], [13], [29].

Scheduling Algorithm for Tests $TST(TQ, S')$

```
Input:  $TQ$, a test queue;
        $S'$, a partial schedule
Output: $S$, a 1-fault-secure schedule
Begin
  For each task $v$ in $TQ$ do
    Find PEs to which no instances of $v$ or its
    predecessors are assigned, and put them into $AP$.
    If ($AP \neq$ NULL) Then
      For each PE $p$ in $AP$ do
        $ST[p]$ := the earliest start time of the test $\tau$
        on $p$.
      End_For
      $p_t$ := the PE where $ST[p_t]$ is the smallest
      Schedule $\tau$ to $p_t$ at time $ST[p_t]$
    Else
      For each PE $p$ in $P$ do
        $ST[p]$ := the earliest start time of the test $\tau_1$
        on $p$.
      End_For
      $p_{t_1}$ := the PE whose $ST[p_{t_1}]$ is the smallest
      Schedule $\tau_1$ to $p_{t_1}$ at time $ST[p_{t_1}]$
      For each PE $p$ in $P - \{p_{t_1}\}$ do
        $ST[p]$ := the earliest start time of the test $\tau_2$
        on $p$.
      End_For
      $p_{t_2}$ := the PE whose $ST[p_{t_2}]$ is the smallest
      Schedule $\tau_2$ to $p_{t_2}$ at time $ST[p_{t_2}]$
    End_If
  End_For
End
```

3.3 Time Complexity

The complexity of task level and height calculation is $O(|E|)$, where $|E|$ denotes the number of edges in the task graph. Each instance of a task is scheduled by applying Procedure TDP to n PEs both in Step 1 and in Step 2 of the Basic algorithm. The computational complexity of Procedure TDP is known to be $O(|V|^3)$ [19], [20], where $|V|$ denotes the number of tasks in the task graph. Therefore, the complexity of scheduling one instance is $O(n|V|^3)$. When calculating the start time of an instance on each PE, Algorithms $FSAE$ and $FSBE$ check whether its predecessors need to be tested or not. The complexity of this checking is $O(n|V|^2)$ for both algorithms. Also, the computational complexity of scheduling of one test is $O(|V|^2)$. Since $|E| < |V|^2$ and the number of tasks is $|V|$, the complexity of the algorithms is $O(|V|^4)$, given that n is fixed.

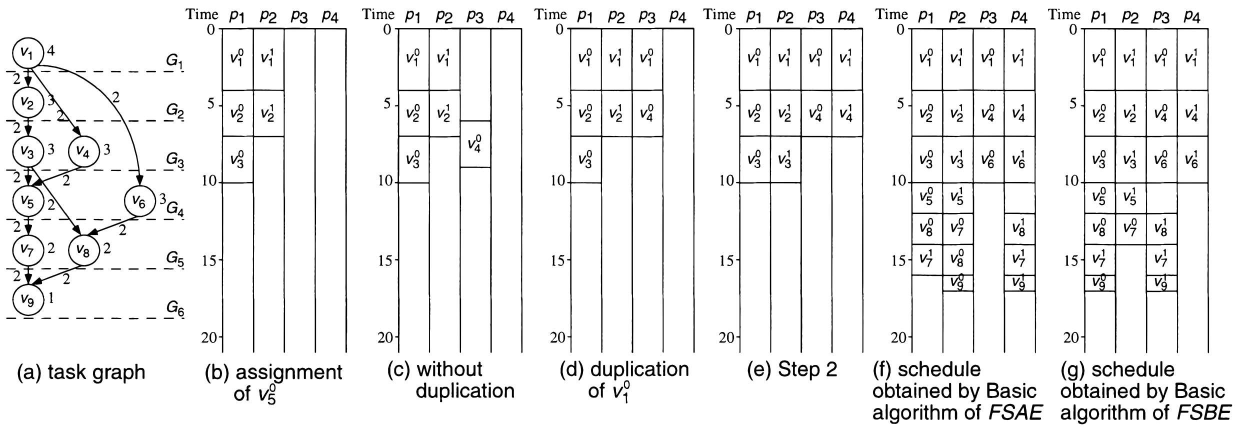

(a) task graph
(b) assignment of v_5^0
(c) without duplication
(d) duplication of v_1^0
(e) Step 2
(f) schedule obtained by Basic algorithm of FSAE
(g) schedule obtained by Basic algorithm of FSBE

Fig. 3 Illustrative example of basic algorithm.

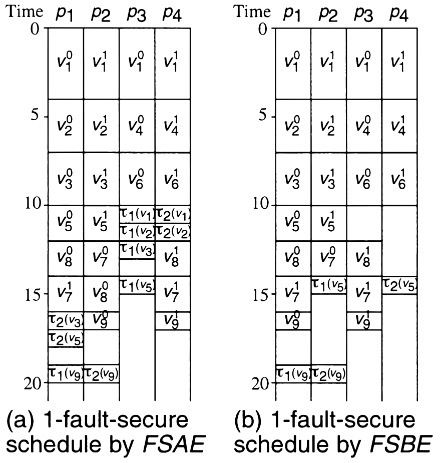

(a) 1-fault-secure schedule by FSAE
(b) 1-fault-secure schedule by FSBE

Fig. 4 Schedules obtained.

3.4 Illustrative Example

Figures 3 and 4 illustrate how Algorithms *FSAE* and *FSBE* work. In this example, we assume that the number of PEs, n, is four and that the task graph shown in Fig. 3(a) is given. The set of tasks is partitioned into six task groups G_1, G_2, ..., G_6. Tasks in each task group are ordered according to their priorities as follows.

G_1: v_1	G_2: v_2	G_3: v_3, v_4
G_4: v_5, v_6	G_5: v_7, v_8	G_6: v_9

These task groups are ordered according to their heights. Then the Basic algorithm is applied to each task group in order. The task group whose height is the largest is selected first.

Now suppose that task groups G_1 and G_2 have been scheduled. Then the Basic algorithm is applied to G_3. In Step 1, each task in G_3 is scheduled, and its instance is tagged with "0". This is done by applying Procedure *TDP* to each PE. For example, an instance of v_4, which is indicated by v_4^0 in Fig. 3, is scheduled as follows. As shown in Fig. 3 (b), an instance of v_3 (indicated by v_3^0) has already been assigned to p_1. It can be seen that the start times of v_4^0 on p_1 and on p_2 are 10 and 7, respectively. The start time of v_4^0 would be 6 on p_3 if no instances were duplicated, as shown in Fig. 3 (c) (Note that v_4 must receive necessary data from v_1). In order to improve the start time of v_4^0, *TDP* applies *task duplication*. Figures 3 (c) and (d) illustrate the concept of task duplication. In this case, *TDP* duplicates v_1 and schedules another instance to p_3 at time 0 (All instances generated in Step 1 are tagged with "0"). As a result of this duplication, v_4^0 can receive necessary data directly from v_1 without any communication delay, and the start time of v_4^0 on p_3 becomes 4. As a result, p_3 can start execution of v_4^0 earlier than p_1 and p_2. Therefore, v_4^0 is scheduled to p_3 as shown in Fig. 3 (d).

In Step 2, each task in G_3 is duplicated and scheduled to one of the n PEs other than the PE to which its instance is already scheduled. For example, since an instance of v_4 (v_4^0) is already scheduled to p_3 in Step 1, *TDP* is applied to p_1, p_2, and p_4. As a result, an instance tagged with "1" (v_4^1) is scheduled to p_4 (All instances generated in Step 2 are tagged with "1"). Similarly, each remaining task is scheduled so as to be executed on two different PEs as shown in Fig. 3 (e).

The Basic algorithm is applied to the remaining task groups G_4, G_5 and G_6. As a result, Algorithms *FSAE* and *FSBE* generate the schedules shown in Figs. 3 (f) and (g), respectively. The difference of the two schedules results from the fact that when more than one PE can run a task earliest among all the PEs, different PEs may be selected by the two algorithms. As stated in Sect. 3.2.2, in such a case, a PE is selected such that the number of tasks that need to be tested is minimized.

When *TDP* calculates the start time of an instance on a PE, both algorithms check whether the predecessors of the instance need to be tested or not, as explained in Sect. 3.2.3. For example, when an instance of v_7 with version number "0" (v_7^0) is scheduled to p_2, *FSAE* decides to test four tasks; namely, v_1, v_2, v_3

and v_5, whereas $FSBE$ decides to test only v_5. In each of the two algorithms' schedules, v_7^0 receives data directly from v_5^1, whose version number is different from v_7^0 (Condition (i) in Sect. 3.2.3). $FSAE$ further decides to test v_1, v_2, v_3, which are predecessors of v_7, because the instances of these tasks that are tagged with "1" are already assigned to p_2 (v_1^1, v_2^1, v_3^1) (Condition (ii) in Sect. 3.2.3). In the schedule for $FSBE$, on the other hand, the instances v_1^1, v_2^1 and v_3^1 on p_2 do not need to be tested. In this case, TQ becomes as follows.

$$FSAE: \quad TQ = \{v_1, v_2, v_3, v_5, v_9\}$$
$$FSBF: \quad TQ = \{v_5, v_9\}$$

Finally, tests for the tasks in TQ are scheduled.

In this example, since v_1, which is a common predecessor to the tasks in TQ, is assigned to all PEs, every test is duplicated and assigned to two distinct PEs. As a result, 1-fault-secure schedules are obtained by $FSAE$ and $FSBE$ as shown in Figs. 4 (a) and (b), respectively.

4. Correctness Proofs of Algorithms

In this section, we prove that the three algorithms we proposed in Sect. 3 generate 1-fault-secure schedules correctly under their corresponding error models. The correctness proof of Algorithm STR is presented in Sect. 4.1. The proofs of $FSAE$ and $FSBE$ are presented in Sects. 4.2 and 4.3, respectively.

In the following proof, we let S denote a schedule generated by any of the algorithms. As in [12], [13], we introduce an MVC_DAG $G' = (V', E')$ for S, where V' is the set of nodes and E' is the set of edges. G' represents the state of data exchanges between instances in S. Thus G' is unique to S. Each node represents an instance of a task in S. If an instance s' receives necessary data from another instance s, then there is an edge from s to s'. If there exists a path from s' to s, we call s' an *ancestor* of s. Figure 5 (c) shows an example of an MVC_DAG for the schedule in Fig. 5 (b).

First, we show some lemmas that are common to all the algorithms.

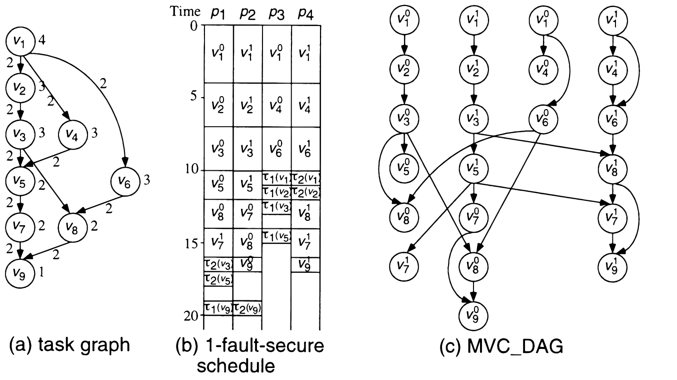


Fig. 5 Example of MVC_DAG.

Lemma 1: For each task $v \in V$, there exists in S a pair of instances of v that have different version numbers ("0" and "1") and are assigned to different PEs.
Proof: It is clear from the definition of the algorithms. □

Lemma 2: For every fault pattern P' and for every interpretation I consistent with P', if an instance s is not contained in P' and $Label_I(s) \neq$ "c," then there is an ancestor t of s such that $t \in P'$ and $Label_I(\alpha) \neq$ "c" for every instance α on a path from t to s.
Proof: Due to Condition (A) in Definition 2, there exists at least one instance α in $D(s)$ such that $Label_I(\alpha) \neq$ "c." Since this argument can also apply to α if $\alpha \notin P'$, the lemma follows. □

Lemma 3: For every fault pattern P' and for every interpretation I consistent with P', if a task v is tested in S, then the following conditions hold:
Case 1: If there exists only one instance t of the test in S, the following two conditions hold for every pair of instances s and s' of v, where s and s' have different version numbers.

1. if $Label_I(s) \neq Label_I(s')$, then the test reliably reports "*not equal*", i.e., $Label_I(t) =$ "n."
2. if the outcome of the test is unreliable, then $Label_I(s) = Label_I(s') =$ "c."

Case 2: If there are two instances of the test in S, the following condition holds for every pair of instances s and s' of v, where s and s' have different version numbers.

1. if $Label_I(s) \neq Label_I(s')$, then one of the two instances of the test, t_1 or t_2 reliably reports "*not equal*," i.e., $Label_I(t_1) =$ "n" or $Label_I(t_2) =$ "n."

Proof: If $Label_I(s) \neq Label_I(s')$, then by Lemma 2 there exists at least one ancestor of s or s' in P'. In Case 1, the algorithms never assign the test to PEs where s, s', or their ancestors have been scheduled. Therefore, $t \notin P'$, that is, the outcome of t is reliable. Consequently, t reliably reports "*not equal*." If t is not reliable, i.e., $t \in P'$, then for the same reason mentioned above, the labels of all the instances of v and its ancestors must be "c." That is, $Label_I(s) = Label_I(s') =$ "c."

In Case 2, there are two instances of the test in S. Each of them has been assigned to a different PE. Since only one PE is assumed to be faulty, it is obvious that either $t_1 \notin P'$ or $t_2 \notin P'$. Therefore, if $Label_I(s) \neq Label_I(s')$, then either t_1 or t_2 reliably reports "*not equal*." □

4.1 Algorithm STR

Since Algorithm STR duplicates a whole schedule for $n/2$ PEs, all instances assigned to each PE are tagged with the same version number. Again, we assume a single faulty PE. From the definition of a fault pattern, therefore, we get the following lemma.

Lemma 4: Given a fault pattern P', every instance of tasks in P' is tagged with the same version number, "0" or "1."

Theorem 1: Schedules that Algorithm STR generates are 1-fault-secure under the redeemable error model.

Proof: As mentioned before, each instance tagged with "i" ($i = 0, 1$) never receives any data from instances tagged with "$1 - i$." By Lemma 4, therefore, every task instance that has a label other than "c" is tagged with a common version number. That is, for any output task v, either $Label_I(s) = Label_I(s')$ ="c" or $Label_I(s) \neq Label_I(s')$ clearly holds where s and s' are the two instances of v. Since every output task is tested, by Lemma 3, schedules generated by STR are 1-fault-secure under the redeemable error model. □

4.2 Algorithm $FSAE$

Let S be a schedule generated by $FSAE$.

Lemma 5: Let s and s' be two instances of a task v, each with a different version number, and let $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \ldots, \alpha'_r)$. For every fault pattern P' and for every interpretation I consistent with P' under the Type A error model, if $Label_I(s) = Label_I(s') \neq$ "c," then the following conditions hold.
Case 1: If $s \notin P'$ and $s' \notin P'$, then $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q ($1 \leq q \leq r$) and there exists at least one p ($1 \leq p \leq r$) such that $Label_I(\alpha_p) = Label_I(\alpha'_p) \neq$ "c."
Case 2: If $s \in P'$ and $s' \notin P'$, then $Label_I(\alpha'_p) \neq$ "c" holds for some p ($1 \leq p \leq r$) (Note that $s, s' \in P'$ never holds by Lemma 1).

Proof: In Case 1, from Conditions (A), (D) and (E) in Definition 2, it is clear that $Label_I(s) = Label_I(s') \neq$ "c" if and only if the condition described above holds. In Case 2, it is clear from Condition (A) in Definition 2 that if $Label_I(s') \neq$ "c," then $Label_I(\alpha'_p) \neq$ "c" for some p ($1 \leq p \leq r$). □

Lemma 6: Let s and s' be two instances of a task v, each with a different version number. For every fault pattern P' and for every interpretation I consistent with P' under the Type A error model, if $Label_I(s) = Label_I(s') \neq$ "c," then there exists an instance t of some test in S such that $Label_I(t) =$ "n."

Proof: We prove this by induction.

[Base Step] From the definition of height, no tasks in the task group G_1 have immediate predecessors. By Lemma 1, therefore, if $v \in G_1$, then $Label_I(s) = Label_I(s') =$ "c" or $Label_I(s) \neq Label_I(s')$ holds.

Next, suppose $v \in G_2$ and $Label_I(s) = Label_I(s') \neq$ "c." When $s, s' \notin P'$, by Lemma 5, there is an instance α such that $Label_I(\alpha) \neq c$ and α is in both $D(s)$ and $D(s')$. In this case, whichever version number α is tagged with, $FSAE$ guarantees that the task corresponding to α is tested. Lemma 3 also ensures that the test for α reliably reports "*not equal*." When $s \in P'$ and $s' \notin P'$, s' receives data from an instance assigned to the same PE as s. Due to the rule of assigning tests, a test is assigned for this instance, and by Lemma 3 it reliably reports "*not equal*."

[Induction Step] Assume that the lemma holds if $v \in G_1 \cup G_2 \cup \cdots \cup G_k (k \geq 2)$. Now suppose $v \in G_{k+1}$ and $Label_I(s) = Label_I(s') \neq$ "c." Let $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \ldots, \alpha'_r)$. Then two cases must be considered:
Case 1: [$s \notin P'$ and $s' \notin P'$]
By Lemma 5, $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q ($1 \leq q \leq r$) and $Label_I(\alpha_p) = Label_I(\alpha'_p) \neq$ "c" for some p ($1 \leq p \leq r$). There are two cases to be considered.

Case 1-1: [α_p and α'_p have different version numbers]
In this case, the task corresponding to α_p and α'_p is in $G_j (j \leq k)$. Due to the assumption, there is a test that reports "*not equal*."

Case 1-2: [α_p and α'_p have the same version number "i"]
In this case, either s or s' receives data from an instance that is tagged with a version number different to i. Then $FSAE$ guarantees testing of the task corresponding to α_p and α'_p. By Lemma 1, there is another instance of this task, say α''_p, with a version number "$i - 1$." If $Label_I(\alpha_p) = Label_I(\alpha'_p) = Label_I(\alpha''_p) \neq$ "c," then due to the assumption, there is a test that reports "*not equal*." If $Label_I(\alpha_p) = Label_I(\alpha'_p) \neq Label_I(\alpha''_p)$, then also, by lemma 3, there is a test that reports "*not equal*."

Case 2: [$s \in P'$ and $s' \notin P'$]
Let i denote the version number of s'. Since $Label_I(s') \neq$ "c," from Lemma 2 there exists an ancestor x of s' in G' such that $x \in P'$ and all instances on a path π from x to s' have labels other than "c." There are two cases to be considered:

Case 2-1: [x is tagged with "i"]
In this case, there exist on π two instances, y tagged with "i" and z tagged with "$1-i$," such that an edge from y to z exists (y may be x itself). Since z receives data from y and their version numbers are different, the task corresponding to y is tested. Let y' be an instance of the same task as y that has version number "$1 - i$." If $Label_I(y) \neq Label_I(y')$, then the test reliably reports "*not equal*" (Lemma 3). If $Label_I(y) = Label_I(y') \neq$ "c," then due to the assumption, there is some other test that reports "*not equal*."

Case 2-2: [x is tagged with "$1 - i$"]
Since $s \in P'$, x is assigned to the same PE as s. When s is assigned to the PE, $FSAE$ de-

cides to test x because x is tagged with a different version number from that of s. Let x' be an instance of the same task as x that has version number "i." If $Label_I(x) \neq Label_I(x')$, then the test reliably reports "*not equal*" (Lemma 3). If $Label_I(x) = Label_I(x') \neq$ "c," then due to the assumption, there is some other test that reports "*not equal*."

Thus, in both Case 1 and Case 2, a test exists that reports "*not equal*." Hence the lemma holds for any $v \in G_{k+1}$, and the lemma follows. □

Finally, we arrive at the following theorem.

Theorem 2: Schedules $FSAE$ generates are 1-fault-secure under the Type A error model.

Proof: Suppose that an instance of an output task v is labelled with an erroneous value. Then, by Lemma 6, there is either a test that reports "not equal," or another instance of v that is labelled with "c." In the latter case, by Lemma 3, there is a test that reliably reports "*not equal*," since all output tasks are tested. Thus the theorem follows from the definition of a 1-fault-secure schedule. □

4.3 Algorithm $FSBE$

Let S be a schedule generated by $FSBE$. Since Conditions (A), (D) and (E) are met in the Type B error model, we get the following lemma.

Lemma 7: Let s and s' be two instances of a task v, each with a different version number, and let $D(s) = (\alpha_1, \alpha_2, \ldots, \alpha_r)$ and $D(s') = (\alpha'_1, \alpha'_2, \ldots, \alpha'_r)$. For every fault pattern P' and for every interpretation I consistent with P' under the Type B error model, if $Label_I(s) = Label_I(s') \neq$ "c," then the following conditions hold.

Case 1: If $s \notin P'$ and $s' \notin P'$, then $Label_I(\alpha_q) = Label_I(\alpha'_q)$ for all q $(1 \leq q \leq r)$ and there exists at least one p $(1 \leq p \leq r)$ such that $Label_I(\alpha_p) = Label_I(\alpha'_p) \neq$ "c."

Case 2: If $s \in P'$ and $s' \notin P'$, then $Label_I(\alpha'_p) \neq$ "c" holds for some p $(1 \leq p \leq r)$.

Lemma 8: Let s and s' be two instances of a task v, each with a different version number. For every fault pattern P' and for every interpretation I consistent with P' under the Type B error model, if $Label_I(s) = Label_I(s') \neq$ "c," then there exists an instance t of some test in S such that $Label_I(t) =$ "n."

Proof: The proof is omitted as it is similar to the one for Lemma 6. The complete proof is presented in [14]. □

Theorem 3: Schedules $FSBE$ generates are 1-fault-secure under Type B error model.

Proof: Suppose that an instance of an output task v is labelled with an erroneous value. Then, by Lemma 8, there is a test that reports "not equal," or there is another instance of v that is labelled with "c." Also in the latter case, there is a test that reliably reports "*not equal*" by Lemma 3, since all output tasks are tested. Thus the theorem follows from the definition of 1-fault-secure schedules. □

5. Experimental Evaluation

By using a large number of task graphs as a work load, we performed simulations for comparison studies of the three proposed algorithms with respect to schedule length. In this section, we present the results and discuss the respective effects of the three error models on schedule length.

5.1 Simulation Environment

In the simulation studies, we used task graphs for two practical parallel computations: Gaussian elimination [21] and LU-decomposition [22]. These task graphs can be characterized by the size of the input matrix because the number of tasks and edges in the task graph depends on the size. For example, the task graph for Gaussian elimination shown in Fig. 1 (a) is for a matrix of size 3. The number of nodes in these task graphs is roughly $O(N^2)$ where N is the size of matrix. We varied the matrix sizes so that the graph sizes ranged from about 100 to 400 nodes.

The *communication-to-computation ratio* (*ccr*) is defined as follows [1], [21]:

$$ccr = \frac{\text{average communication delay between tasks}}{\text{average execution time of tasks}}$$

For each task graph size, we generated six different graphs for *ccr* values of 0.1, 0.5, 1.0, 2.0, 5.0 and 10.0 by varying communication delays.

In practice, the value of *ccr* varies over a very large range, depending on architectures of multiprocessors and the granularity of tasks. For finest-grain cases, the value can exceed 1000 even in modern parallel machines, such as Fujitsu AP-1000 or NEC Cenju-3 [3]. However, such fine-grain tasks are not usually used in practice, since the number of tasks becomes so huge that scheduling cannot be completed in a reasonable time. Thus we exclude such extreme cases in the simulation.

We assume that the execution time required by every test is identical, and that the communication delay needed for each test to receive data from the participating instances is identical. For each task graph, the execution time of a test is set to the smallest execution time among all tasks, and the communication cost between a test and tasks is set to the average communication delay between tasks.

As a baseline, we used the finish time of a (non-fault-secure) schedule generated by DSH. All results presented in this section are normalized to this length. In the studies, we considered two cases: the number of

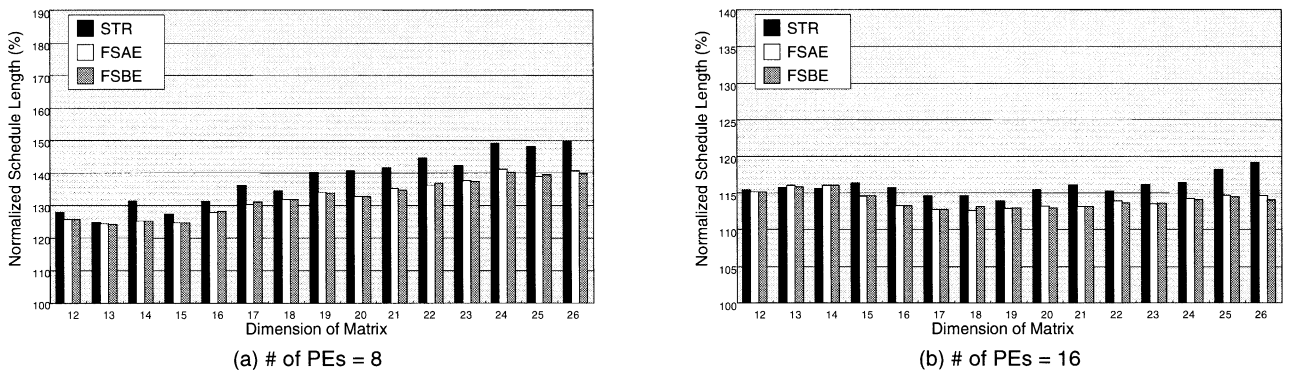


Fig. 6 Results for Gaussian elimination task graphs with $ccr = 5.0$.

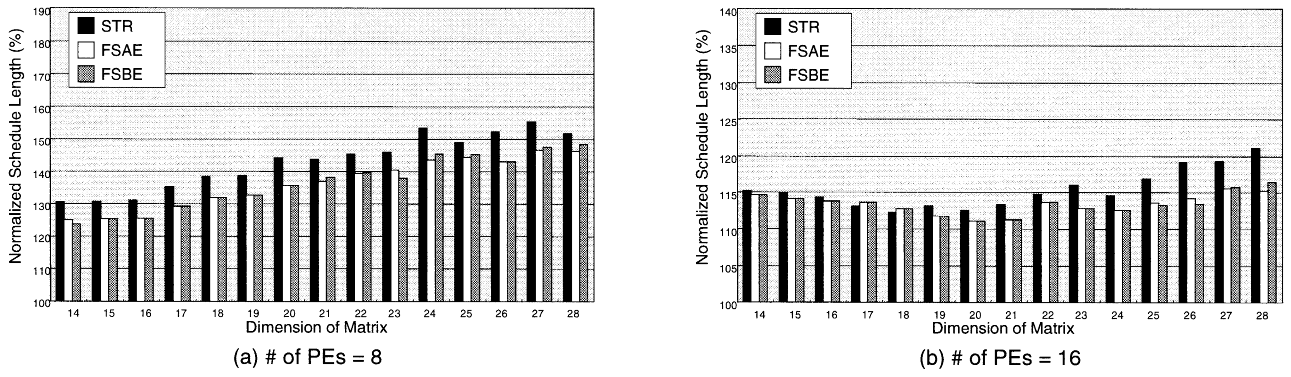


Fig. 7 Results for LU-decomposition task graphs with $ccr = 5.0$.

PEs $n = 8$ and $n = 16$.

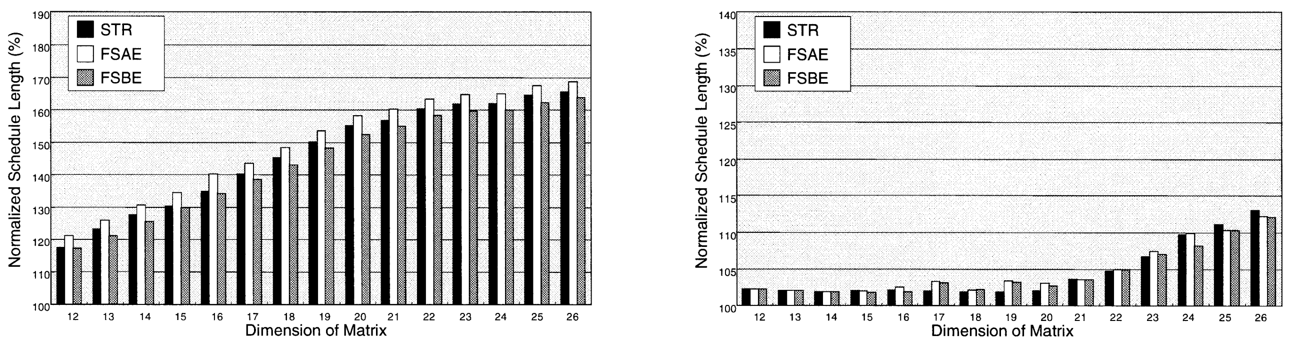
5.2 Evaluation Results

Figures 6 and 7 show the simulation results for Gaussian elimination and LU-decomposition task graphs, respectively. The value of ccr is fixed to 5.0, and the matrix size is varied so that the number of tasks in the corresponding task graph ranges from 100 to 400. The results show that both $FSAE$ and $FSBE$ outperform STR. As the matrix size increases, the difference between the performance of $FSAE(FSBE)$ and that of STR increases. The following reason is conjectured. In general, as the size of the task graph increases, its parallelism also increases (here, parallelism means the maximum number of tasks that can be executed in parallel at a time). We can exploit the parallelism only if we have a sufficient number of PEs. In this simulation, the number of PEs n is fixed regardless of the size of task graph. Therefore, as the matrix size increases, $FSAE(FSBE)$ can extract more parallelism than STR because STR can essentially use only $n/2$ PEs.

In the case of $ccr = 5.0$, the difference in performance between $FSAE$ and $FSBE$ is not clear. The number of tests $FSAE$ requires is usually larger than that of $FSBE$. For example, for a Gaussian elimination task graph with matrix size 24, we found that the number of tests that $FSAE$ required was 335, whereas that of $FSBE$ was only 164. However, when communication overhead is large, the number of idle time slots between tasks (i.e., the time slots available for scheduling tests) for $FSAE$ to avoid a performance degradation.

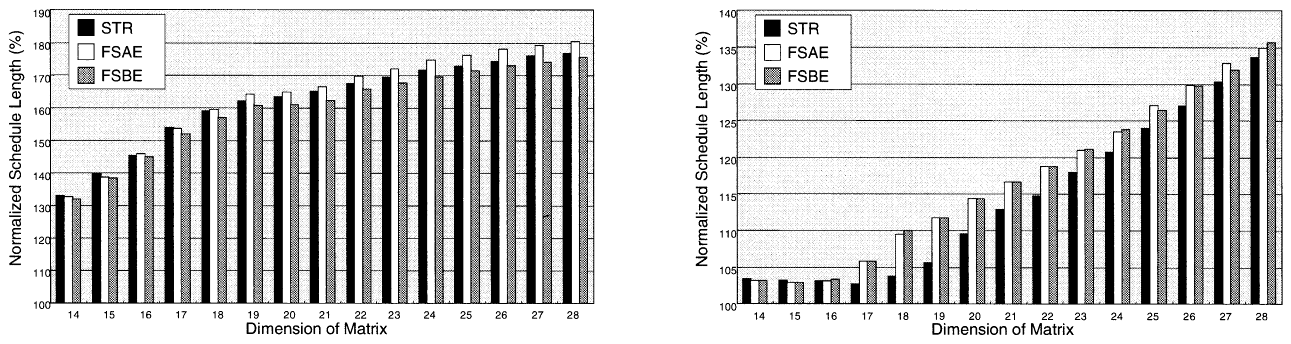
Figures 8 and 9 show, respectively, the simulation results for Gaussian elimination and LU-decomposition task graphs with $ccr = 0.5$. The matrix size is varied in the same way as in the case of $ccr = 5.0$. In the case of $n = 8$, the difference in performance between $FSAE$ and $FSBE$ is observed more clearly. In addition, $FSAE$ has worse performance than STR. This is because, as mentioned above, when communication overhead is small, the number of idle time slots is small.

Figures 10 and 11 show the simulation results when the matrix size is 24. In this simulation, we varied the value of ccr from 0.1 to 10.0. In both kinds of task graphs, when the value of ccr is small (e.g., $ccr < 1.0$), STR shows better performance than $FSAE$. Additionally, as the value of ccr is decreased, $FSBE$ exhibits better performance than $FSAE$. In most cases, the number of tests in a schedule obtained by STR is much smaller than $FSAE$ or $FSBE$ because STR requires tests for the output tasks only. Note that as communication delays decrease, the amount of idle time, which can be used to schedule tests by $FSAE$ or $FSBE$, decreases. As a result, $FSAE$ and $FSBE$ have worse



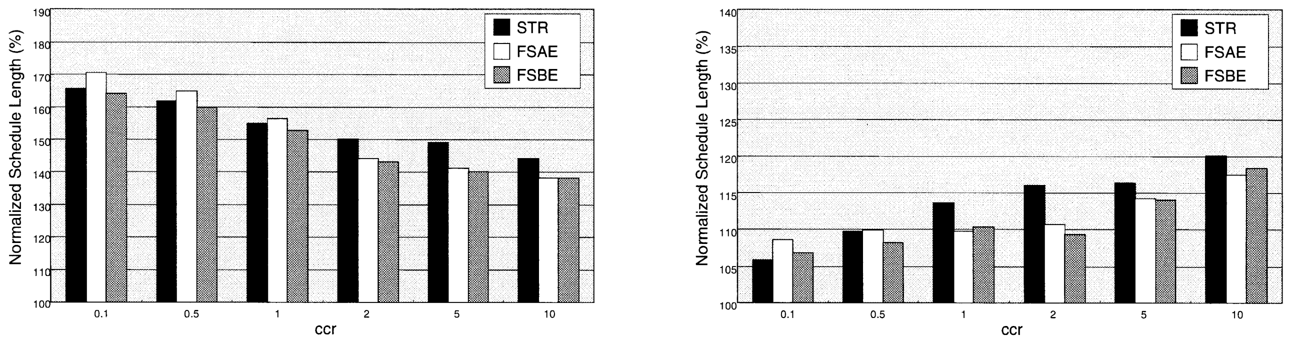
(a) # of PEs = 8 (b) # of PEs = 16

Fig. 8 Results for Gaussian elimination task graphs with *ccr* = 0.5.



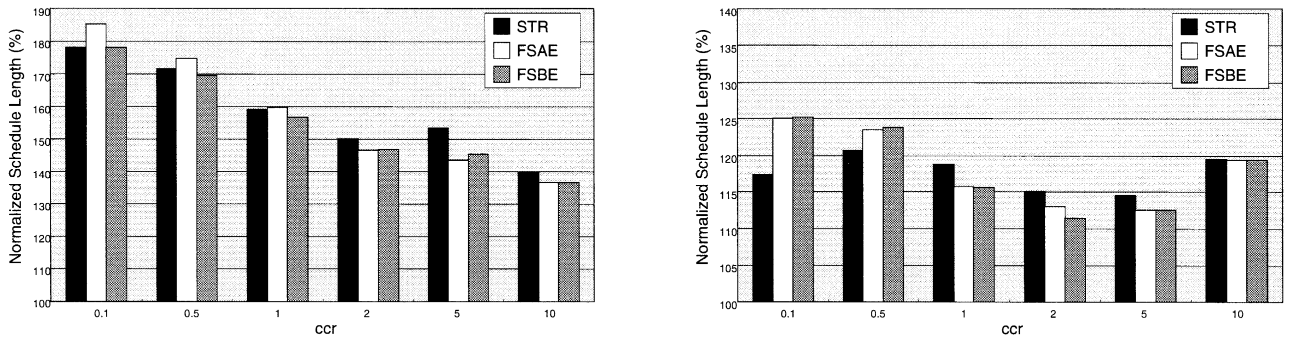
(a) # of PEs = 8 (b) # of PEs = 16

Fig. 9 Results for LU-decomposition task graphs with *ccr* = 0.5.



(a) # of PEs = 8 (b) # of PEs = 16

Fig. 10 Results for Gaussian elimination task graphs.



(a) # of PEs = 8 (b) # of PEs = 16

Fig. 11 Results for LU-decomposition task graphs.

performance than STR when ccr is small.

Compared with the non-fault-secure scheduling algorithm DSH, $FSAE$ and $FSBE$ achieved 1-fault-security at the cost of a small increase in schedule length. For example, in the case where $n = 16$ and $ccr \geq 1$, the algorithms achieved 1-fault-security with less than 20% overhead. (Note that each result is normalized to the schedule length of DSH.)

All schedules generated by the three proposed algorithms are 1-fault-secure. However, since the fault models that are assumed by these algorithms are different, the degrees of fault security achieved are also different. In the following subsection, we discuss the results from the viewpoint of both performance and fault security.

5.3 Discussion

Since the number of tests scheduled by Algorithm $FSBE$ is smaller than that of Algorithm $FSAE$, $FSBE$ has better performance than $FSAE$, particularly when communication overhead is small. When communication overhead is large, $FSAE$ and $FSBE$ show better performance than Algorithm STR, although the performance of $FSBE$ is close to that of $FSAE$. However, in some cases, STR outperforms $FSAE$ and $FSBE$. In this simulation, the execution time of a test is set to the smallest execution time of tasks in the given task graph. It can be considered that as the execution time of a test increases, STR more often outperforms $FSAE$ and $FSBE$.

As defined in Sect. 2.3, the Type B error model is a subclass of the Type A error model, while the Type A error model is a subclass of the redeemable error model. Therefore, schedules that Algorithm $FSAE$ generates are 1-fault-secure under the Type A error model and the Type B error model, whereas schedules that Algorithm STR generates are 1-fault-secure under the redeemable error model, the Type A error model and the Type B error model. Therefore, in situations where the number of idle time slots is not sufficient to assign instances of tests, STR can be chosen even if the redeemable error model is too pessimistic for the required fault-security level.

In the paper, we have restricted our discussions to 1-fault-security. However, when multiple PE failures are considered, the situation becomes much complex, since 1-fault secure schedules may tolerate failures of more than one PE. In the case of multiple PE failures, STR is more robust than the other two algorithms, because faults can be detected as long as failures center on one half of the PEs. (Note that STR duplicates a whole non-fault-tolerant schedule.) With $FSAE$ and $FSBE$, this is not the case, since these algorithms distribute tasks among all PEs irrespective of their version numbers, aiming at shortening schedule length. Thus when fault probability is high or correlated faults [27] can occur, the use of STR is suggested. Although probabilistic analysis is beyond the scope of the paper, it definitely deserves further work.

6. Related Work

In recent years, much research has been developed on methods for achieving fault tolerance in parallel and distributed systems through task scheduling. These methods have received much attention as a cost effective means of high reliability, since no hardware dedicated for fault tolerance is required.

These methods assume a variety of system and fault models. Methods that deal with independent tasks include [6], [7], [10], [23], [28]. Most of them assume that tasks arrive at the system dynamically (exceptions include [6]). Each arriving tasks is replicated and scheduled to more than one processors in order to detect processor failures and/or to produce correct results.

Fault-tolerant scheduling methods that can deal with tasks with constraints are also well studied. These methods assume that tasks have mutual dependency, and the relations are usually specified by a directed graph. In [18], [25], methods for scheduling communicating tasks onto distributed systems are presented.

Scheduling task graphs onto multiprocessors is discussed in [4], [5], [12], [13], [15], [29]. As stated in the introduction, fault-secure scheduling is studied in [4], [12], [13], [29]. The previous research deals with tree-structured task graphs only and does not consider communication overheads. On the other hand, the algorithms proposed in this paper can handle arbitrary task graphs with communication delays.

In [5] and [15], much more benign failure models are assumed than the three error models presented in the paper. More specifically, [5] assumes that fault can be detected by specialized tasks, while *fail-stop processors* [26] are assumed in [15]. Scheduling algorithms proposed in these papers are aimed at masking faults, instead of detecting them.

Other related research includes [11], which proposes an algorithm for scheduling loops in parallel programs to detect faults.

It should be noted that fault tolerance of VLSIs can also be achieved through scheduling in high-level synthesis. Research in this direction is, for example, [8] and [17].

7. Conclusions

In this paper, we introduced three error models (the redeemable, Type A, and Type B error models), and investigated the respective effects of the different error models on the fault security of a multiprocessor schedule. The Type B error model is a subclass of the Type A error model, whereas the Type A error model is a subclass of the redeemable error model.

To achieve the 1-fault-secure property under the redeemable error, Type A, and Type B error models, we proposed algorithms STR, $FSAE$ and $FSBE$, respectively. We proved that these algorithms produce 1-fault-secure schedules with time complexity $O(|V|^4)$, where $|V|$ is the number of tasks in a given task graph.

In order to study the impact of the error models on schedule length, we performed simulation studies. As a result, it was found that in a situation where communication overhead is large ($ccr \geq 1$), $FSAE$ and $FSBE$ outperform STR, but the difference in performance between $FSAE$ and $FSBE$ cannot be seen clearly. On the other hand, when communication overhead is small ($ccr < 1$), STR exhibits better performance than $FSAE$ and $FSBE$ in some cases. In such cases, STR should be chosen even though the redeemable error model is more pessimistic than the required fault-security level. Compared with a non-fault-secure scheduling algorithm DSH, the proposed algorithms achieve 1-fault-security at the cost of a small increase in schedule length.

Acknowledgements

The authors would like to thank Dr. Noriyuki Fujimoto and Mr. Aaron Stokes for their helpful suggestions. The authors also thank the anonymous referees for their helpful comments on how to improve this paper.

References

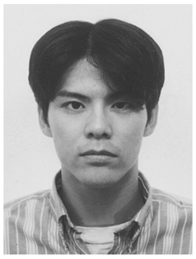
[1] I. Ahmad and Y.-K. Kwok, "A new approach to scheduling parallel programs using task duplication," Proc. International Conference on Parallel Processing, pp.II-47-51, 1994.
[2] I. Ahmad and Y.-K. Kwok, "On exploiting task duplication in parallel program scheduling," IEEE Trans. Parallel and Distributed Systems, vol.9, no.8, pp.872–892, 1998.
[3] T. Baba, T. Hashimoto, N. Fujimoto, and K. Hagihara, "A task scheduling algorithm with consideration to communication property on a distributed memory parallel machine," IEICE Technical Report, COMP98-72, 1999.
[4] P. Barnerjee and J.A. Abraham, "Fault-secure algorithms for multiple processor systems," Proc. 11th International Symposium on Computer Architecture, pp.270–287, 1984.
[5] S. Chabridon and E. Gelenbe, "Failure detection algorithms for a reliable execution of parallel programs," 14th IEEE International Symposium on Reliable Distributed Systems, pp.229–238, 1995.
[6] V. Cherkassky and C.-I.H. Chen, "Redundant task-allocation in multicomputer systems," IEEE Trans. Reliability, vol.41, no.3, pp.336–342, 1992.
[7] A. Dahbura, K.Sabnani, and W. Hery, "Space capacity as a means of fault detection and diagnosis in multiprocessor systems," IEEE Trans. Comput., vol.38, no.6, pp.881–891, 1989.
[8] B.P. Dave and N.K. Jha, "COFTA: Hardware-software co-synthesis of heterogeneous distributed embedded systems," IEEE Trans. Comput., vol.48, no.4, pp.417–441, 1999.
[9] H. El-Rewini, H.H. Ali, and T. Lewis, "Task scheduling in multiprocessing systems," IEEE Comput., vol.28, no.12, pp.27–37, 1995.
[10] S. Ghosh, R. Melhem, and D. Mossé, "Fault-tolerance through scheduling of aperiodic tasks in hard real-rime multiprocessor systems," IEEE Trans. Parallel and Distributed Systems, vol.8, no.3, pp.272–284, 1997.
[11] C. Gong, R. Melhem, and R. Gupta, "Loop transformations for fault detection in regular loops on massively parallel systems," IEEE Trans. Parallel and Distributed Systems, vol.7, no.12, pp.1238–1249, 1996.
[12] D. Gu, D.J. Rosenkrantz, and S.S. Ravi, "Construction and analysis of fault-secure multiprocessor schedules," Proc. 21th IEEE International Symposium on Fault-Tolerant Computing, pp.120–127, 1991.
[13] D. Gu, D.J. Rosenkrantz, and S.S. Ravi, "Fault/error models and their impact on reliable multiprocessor schedules," Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, pp.176–184, 1992.
[14] K. Hashimoto, "Multiprocessor scheduling algorithms for high reliability," Ph.D. dissertation, Dept. of Informatics and Mathematical Science, Osaka University, 2000.
[15] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "A multiprocessor scheduling algorithm for low overhead fault-tolerance," Proc. 17th IEEE International Symposium on Reliable Distributed Systems, pp.186–194, 1998.
[16] B.W. Johnson, Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wesley Publishing Company, 1989.
[17] R. Karri and A. Orailoglu, "Time-constrained scheduling during high-level synthesis of fault-secure VLSI digital signal processors," IEEE Trans. Reliability, vol.45, no.3, pp.404–412, 1996.
[18] S. Kartik and C. Siva Ram Murthy, "Task allocation algorithms for maximizing reliability of distributed computing systems," IEEE Trans. Comput., vol.46, no.6, pp.719–724, 1997.
[19] B. Kruatrachue, "Static task scheduling and grain packing in parallel processing systems," Ph.D. dissertation, Electrical and Computer Eng. Dept., Oregon State Univ., Corvallis, 1987.
[20] B. Kruatrachue and T.G. Lewis, "Static task scheduling and grain packing in parallel processing systems," IEEE Software, vol.5, no.1, pp.22–32, 1988.
[21] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," IEEE Trans. Parallel and Distributed Systes, vol.7, no.5, pp.506–521, 1996.
[22] R.E. Lord, J.S. Kowalik, and S.P. Kumar, "Solving linear algebraic equations on an MIMD computer," J. ACM, vol.30, no.1, pp.103–117, 1983.
[23] G. Manimaran and C. Siva Ram Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-rime systems and its analysis," IEEE Trans. Parallel and Distributed Systems, vol.9, no.11, pp.1137–1152, 1998.
[24] K.R. Pattipati, T. Kurien, R.-T. Lee, and P.B. Luh, "On mapping a tracking algorithm onto parallel processors," IEEE Trans. Aerosp. & Electron. Syst., vol.26, no.5, pp.774–791, 1990.
[25] S.M. Shatz, J.-P. Wang, and M. Goto, "Task allocation for maximizing reliability of distributed computer systems," IEEE Trans. Comput., vol.41, no.9, pp.1156–1168, 1992.
[26] R.D. Schlichting and F.B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," ACM Trans. Computer Systems, vol.1, no.3, pp.222–238, 1983.
[27] D. Tang and R.K. Iyer, "Analysis and modeling of correlated failures in multicomputer systems," IEEE Trans. Comput., vol.41, no.5, pp.567–577, 1992.
[28] S. Tridandapani, A.K. Somani, and U.R. Sandadi, "Low overhead multiprocessor allocation strategies exploiting sys-

tem spare capacity for fault-detection and location," IEEE Trans. Comput., vol.44, no.7, pp.865–877, 1995.
[29] J. Wu, E.B. Fernandez, and D. Dai, "Optimal fault-secure scheduling," Comput. J., vol.41, no.4, pp.208–222, 1998.

Appendix: *TDP* (Task Duplication Procedure)

```
Procedure $TDP(v_i,P_c,DTlst)$
Input:  $v_i$, assigned task;
        $P_c$, assigned PE candidate;
        $DTlst$, a list of duplicated tasks;
Output: $ST_i$, $v_i$'s start time;
Begin
  $DTlst' := DTlst$
  Repeat
    $DTlst := DTlst'$
    $flag :=$ UNSUCCESSFUL
    $ST_i :=$ Start_Time($v_i$,$P_c$,$DTlst'$)
    /*$ST_i$ is the start time of $v_i$ on $P_c$, given that
     tasks in $DTlst'$ are duplicated.*/
    IF ($v_i$ has predecessors) Then
        $LIP :=$ Find_LIP($v_i$,$P_c$,$DTlst'$)
        /*$LIP$ is an immediate predecessor of $v_i$ that
         directly causes the start time of $v_i$ (i.e., $ST_i$)
         [19].*/
        If ($LIP$ is not assigned to $P_c$) and
          ($LIP$ is not duplicated in $DTlst'$) Then
            $ST_{LIP} := TDP(LIP,P_c,DTlst')$
            If (Start_Time($v_i$,$P_c$,$DTlst'$)$\leq ST_i$) Then
                $flag :=$ SUCCESSFUL
            End_If
        End_If
    End_If
  Until ($flag =$ UNSUCCESSFUL)
  Insert $v_i$ into $DTlst$ at $ST$
End
```

Koji Hashimoto received the M.E. and Ph.D. degrees in computer engineering from Osaka University in 1997 and 2000, respectively. He is currently with Hitachi Laboratory of Hitachi Ltd. He is a member of IEEE.



Tatsuhiro Tsuchiya received the M.E. and Ph.D. degrees in computer engineering from Osaka University in 1995 and 1998, respectively. He is currently an assistant professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests are in the areas of automatic verification and distributed computing. He is a member of IEEE.

Tohru Kikuno received M.E. and Ph.D. degrees from Osaka University in 1972 and 1975, respectively. He was with Hiroshima University from 1975 to 1987. Since 1990, he has been a professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests include the quantitative evaluation of software development processes and the analysis and design of fault-tolerant systems. He served as a program co-chair of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98) and the Fifth International Conference on Real-Time Computing Systems and Applications (RTCSA '98).