Vol. 36 No. 9 Transactions of Information Processing Society of Japan Sep. 1995

Regular Paper

A Scheduling Strategy for Tasks with Precedence and Conditional Execution

HIDENORI NAKAZATO,† JANE W. S. LIU †† and TAEWHAN KIM †††

A program in a computer system can be divided into program segments, or tasks. Tasks typically have precedence relations among them. Execution flow control constructs in programming languages, such as IF and CASE statements, cause conditional task execution. In other words, a task may dispatch one sequence of tasks in one condition and the other sequence of tasks in the other condition. When there are such conditional task sequence executions in a time-critical system, we need to consider schedules for all possible combinations of task sequences in order to guarantee the deadlines assigned to the task set. The number of schedules we need to consider increases exponentially with the number of conditional task sequences. In order to solve this difficulty, we adopt a three-stage scheduling strategy which has been proposed for VLSI design. First, the original task precedence relation with conditional execution is transformed into a task precedence relation without conditional execution. Then, the transformed task set is scheduled. Finally, the schedule of the transformed task set is transformed back to a schedule of the original task set. In this paper, several transformation algorithms are proposed for the first-stage. The performance of the algorithms are compared through simulations.

1. Introduction

When segments of programs, which we call *tasks*, are executed on a computer, precedence relations exist among them. The precedence relations can be described by a directed graph called a *precedence graph*. In a precedence graph, nodes correspond to tasks. We denote a task (and a node) by τ_i. If there is an edge from node τ_i to node τ_j, task τ_j cannot start until task τ_i completes. τ_j is called an *immediate successor* of τ_i. Conversely, τ_i is an *immediate predecessor* of τ_j. A node may have more than one outgoing edge. Such multiple outgoing edges can have two meanings. One meaning is that all successors must be executed. The other meaning is that only one of the successors is to be executed. We say that the outgoing edges expresses an *AND constraint* in the former case and an *OR constraint* in the latter case. A node (task) whose outgoing edges express an OR constraint is called a *fork* node (task). **Figure 1** is an example. In this precedence graph, task τ_1 has two immediate successors, and its outgoing edges express an OR constraint. We represent an OR constraint by an arc between edges. With such an OR constraint, either the tasks in the left subgraph or the tasks in the right subgraph are to be executed, but never both. We call each subgraph whose source node is pointed by an edge representing an OR constraint a *conditional branch*. By a source node, we mean a node without predecessors in the subgraph. Task τ_2 has two outgoing edges, and the edges here represent an AND constraint. All immediate successor tasks of τ_2 need to be executed after τ_2's completion in this case.

OR constraints make deterministic scheduling for time-critical systems difficult. Since we do not know which conditional branch will be executed until the execution of each fork task completes, we must consider all possible combinations of tasks that are to be executed. Complexity of scheduling increases exponentially with the number of fork tasks.

Suppose a set of tasks which forms a precedence graph with OR constraints has one deadline. On a single processor system, we can test whether the task set is schedulable before the deadline in polynomial time of the number of tasks. We can transform the problem of determining the schedulability of tasks with the OR constraints into a problem of finding the longest path in an acyclic directed graph. The longest path problem is solvable in polynomial time of the number of tasks.[5),8)] If the task set is executed on a multi-processor system and the task set has resource constraints, the scheduling problem is already NP-Hard without OR constraints.[4),5),12)] The OR constraints impose

† Oki Electric Industry
†† Department of Computer Science University of Illinois
††† Lattice Semiconductor Corporation

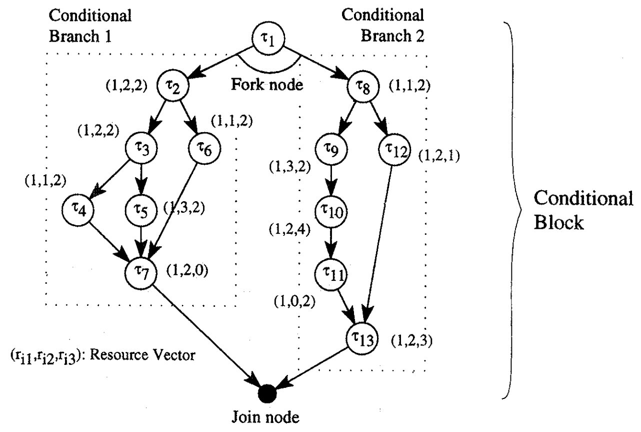


Fig. 1 Example of precedence graph.

additional difficulty on the scheduling problem.

For the multi-processor scheduling with resource constraints and OR constraints, we adopt a three-stage scheduling strategy proposed by Kim et al.[7] In the first stage, a given precedence graph with OR constraints is transformed to a precedence graph without OR constraints. We then schedule the tasks as if their dependencies are defined by the precedence graph without OR constraints. In the last stage, we transform the resultant schedule into a schedule of the original set of tasks. Because of the way the graph without OR constraints is generated, it is always possible for us to do the transformation in the last stage. In this paper, we concentrate on the first stage; we can use any known algorithm for scheduling tasks without OR constraints in the second stage, and the transformation in the third stage is straightforward.

In the first stage, multiple tasks may be combined into a single task, which we call a *union task*. In other words, one node may represent more than one task after the transformation. We call the set of tasks represented by one union task the *original tasks* of the union task. Only one among all the original tasks of each union task will actually execute. Since which task will execute is unknown a priori when the tasks are scheduled, we must allocate the union task the resources required by all of its original tasks. Some of the resources allocated to a union task are wasted at run time when its original task that executes does not need all the allocated resources. On a single processor system, we need not be concerned about wasting resources because the wasted resources cannot be used by other tasks anyway. However, on a multiprocessor system, we should minimize the wasted resources. We may be able to execute more tasks simultaneously by minimizing wasted resources and satisfy deadlines in case of time-critical systems. Minimizing wasted resources is the same as maximizing the usage of resources shared by original tasks combined into a union task. We call these resources shared by original tasks *conditionally shared resources*. One objective of the transformation is to maximize the usage of conditionally shared resources.

By combining more than one original tasks into one union task, we may lengthen the logest path of the precedence graph. Lengthening the longest path may postpone the completion time of the task set. This is not desirable in time-critical systems. We should keep the longest path short through the transformation. This is another objective of the transformation. However, probability to increase the usage of the conditionally shared resources grows if we lengthen the longest path. We need to trade off between the two objectives.

This study is inspired by research on scheduling of computation steps in VLSI design.[3),7),11),13)] Since the target of their algorithms is VLSI design, they assume each unit of execution requires only one resource. In a computer system, tasks may require more than one resource. Our model handles this case. In real-time systems research Gillies and Liu[6] studied AND/OR precedence constraint. However, their OR constraint is on incoming edges of a node while ours is on outgoing edges.

In the next section, we formally define our model of tasks, precedence relations, and resource requirements. In Section 3, we propose a criterion of a good transformation. Section 4 describes several transformation algorithms. A simulation study to evaluate these algorithms and the results on their relative performance are presented in Section 5. Section 6 concludes the paper.

2. System Model

A precedence graph with AND and OR constraints has three types of nodes: *fork* nodes, *join* nodes, and *operation* nodes. A fork node has more than one conditional branch as successors. In other words, a fork node is a node with an OR constraint. The node τ_1 in Fig. 1 is a fork node. The conditional branches of a fork node join together at a join node. The filled node in Fig. 1 is a join node. Join nodes are used for notational convenience and have zero execution time. The set of conditional branches between a fork node and the corresponding join node, together with the join node and the fork node, is called a *conditional block*.

If a programming language construct such as a GOTO statement that can transfer execution flow to an arbitrary position is used in a conditional statement, the join node for the conditional statement may be located far down the precedence graph and conditional blocks may not be properly nested. However, programs can always be written without GOTO statements. We assume that GOTO statements are not used and conditional blocks are properly nested in this paper.

Nodes other than fork nodes and join nodes are called operation nodes. An operation node may have many immediate successors and multiple outgoing edges. All immediate successors of an operation node must be executed. A task that corresponds to a node can be one operation or a sequence of operations as long as the execution time of the task is known. In this paper, we assume tasks represented by fork nodes and operation nodes have unit execution times.

The tasks are not preemptive, i.e., once a task starts execution, the execution cannot be interrupted. In addition to non-preemptive tasks with identical execution times, this model can be applied to characterize tasks that have arbitrary execution times and are preemptive at discrete points. The unit execution time in this model is the time slice in which no preemption is allowed. Then, a task can be split into many unit execution time tasks with straight line precedence constraints.

In addition to the processors, each task may require resources of different types. There are m types of reusable resources including processors. Let ρ_j be a resource of j-th type. Task τ_i requires r_{i1} units of resource ρ_1, r_{i2} units of ρ_2, and so on. The resource requirements of task τ_i are given by its *resource vector* $(r_{i1}, r_{i2}, \cdots, r_{im})$. In Fig. 1, node τ_i is labeled by this vector. While a task accesses a resource, the other tasks cannot use the resource. In other words, a task need exclusive access to a resource in our model. After the task finishes accessing the resource, the other tasks can use it.

3. Transformation

A transformation algorithm transforms a set of conditional branches in a precedence graph into a subgraph without conditional branches. We now describe the underlying transformation process and the objective function to be optimized.

The basic step of the transformation is to select two nodes in different conditional branches in a conditional block and combine the two nodes into one node in the precedence graph. In other words, two original tasks are combined into one union task. The immediate successors of the original tasks are the immediate successors of the union task, and the immediate predecessors of the original tasks become the immediate predecessors of the union task. The resource requirement of the union task is the union of the resources required by the original tasks. By repeating the combining process of tasks, we can combine the conditional branches in each conditional block into one unconditional branch. The transformation starts from the inner-most conditional block and proceeds to the surrounding outer conditional blocks if conditional blocks are nested.

All resources required by a union task may not be used at run time because only one of the original tasks is executed. Resources which are not conditionally shared by all original tasks may be wasted. Some combinations of original tasks may lead to less wasted resources than the others. As an illustrative example, suppose that a system has 2 processors, 2 units of ρ_1, 4 units of ρ_2, and 4 units of ρ_3. If original tasks τ_4 and τ_7 in **Fig. 2** are combined into a union task $\tau_{(4,7)}$, $\tau_{(4,7)}$'s resource vector is $(1, 3, 3)$. Both

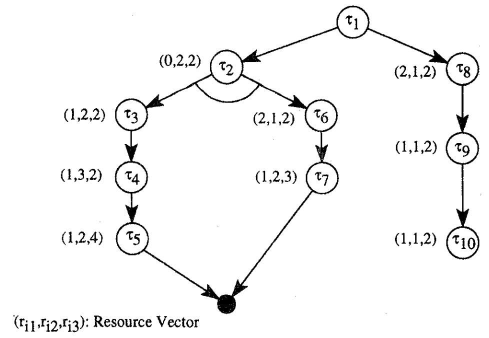


Fig. 2 Conditional resource sharing.

tasks τ_9 and τ_{10}, which are not in this conditional block and have the resource vector (1, 1, 2), cannot be scheduled simultaneously with the conditional block. However, if τ_5 and τ_7 are combined and τ_4 is left alone, τ_9 can be scheduled simultaneously with τ_4 and execution of the right branch ($\tau_8 \rightarrow \tau_9 \rightarrow \tau_{10}$) can be speeded up. In the former transformation, either one of ρ_2 or ρ_3 is always wasted while at most one ρ_3 may be wasted in the latter.

The length of the longest path in a conditional block gives the minimum completion time of the transformed conditional block. Here, by the length of a path in the precedence graph, we mean the number of nodes on the path. The longest path does not include the join node because a join node has zero execution time. For example, the length of the longest path of the conditional block in Fig. 1 is 6, and completing this conditional block takes at least 6 units of time. Combining two nodes may lengthen the longest path. If we combine nodes τ_2 and τ_{10} in Fig. 1, the precedence graph after combining the two tasks is as shown in **Fig. 3** and its longest path length is 7. The dashed arrow in Fig. 3 is a precedence relation exists in the original precedence graph. However, after combining τ_2 and τ_{10}, this relation need not be expressed explicitly because the path $\tau_1 \rightarrow \tau_8 \rightarrow \tau_9 \rightarrow \tau_{(2,10)}$ implicitly expresses the precedence relation.

Suppose that there are two conditional branches with the same longest path and straight line precedence constraints. If we are not allowed to lengthen the longest path, there is only one way to combine the two conditional branches. If we relax this restriction, we have more choices in order to reduce wasted resources. Lengthening the longest path, and thus postponing the completion time of the

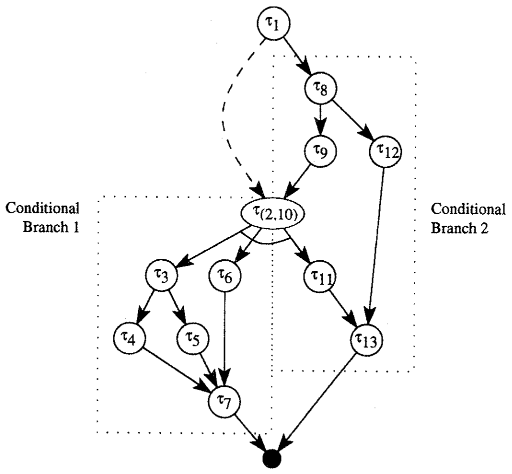


Fig. 3 Example of longest path length change.

conditional block, is not desirable especially in time-critical systems. However, by reducing the wasted resources, concurrently executing tasks may finish early. The transformation algorithms described in Section 4 will attempt to trade off between the wasted resources and the longest path length.

We evaluate the goodness of combining two tasks τ_x and τ_y with resource vectors (r_{x1}, r_{x2}, $\cdots$, r_{xm}) and (r_{y1}, r_{y2}, $\cdots$, r_{ym}) by the cost function

$$R_{(x,y)} = \alpha_1 P_{(x,y)1} + \alpha_2 P_{(x,y)2} + \cdots + \alpha_m P_{(x,y)m},$$

where $P_{(x,y)j} = \min(r_{xj}, r_{yj}) = P_{(y,x)j}$ and α_is are weights. If all α_js are equal to 1, $R_{(x,y)}$ is the total number of resources conditionally shared by τ_x and τ_y. α_js are assigned by the system designer. The more efficiently resource type ρ_j should be used, the larger α_j is. τ_x and τ_y can be not only original tasks but also union tasks.

The *resource sharing index* of a transformation is given by the sum of $R_{(x,y)}$s, i.e.,

$$R = \sum_{(\tau_x, \tau_y) \in \Phi} R_{(x,y)},$$

where $\Phi = \{(\tau_x, \tau_y) \mid \tau_x \text{ and } \tau_y \text{ are combined}\}$ is the set of all union tasks in the resultant graph. The more efficiently resources are used, the larger R is. Let e be a parameter which is the permitted maximum length of the longest path in the resultant graph. The value e affects to the latest acceptable time to complete execution of the conditional block. Since lengthening the longest path increases the minimum completion time of the conditional block, we measure the goodness of the transformation with R/e. We call R/e the *effectiveness index* of the transformation. A small longest path length or a large resource sharing index means large R/e and a good transformation has a large R/e. The algorithms described in the next section attempt to find transformations with good effectiveness indices for an acceptable value of e.

4. Transformation Algorithms

In our algorithms, for each chosen value e, tasks in the conditional branches are iteratively combined while limiting the longest path length to no longer than e. For a given value e, the time interval between the earliest time and the latest time, relative to the completion time of the fork node, when a task can be scheduled is referred to as the *time frame* of the task. The time frame of a task can be determined by using the ASAP (as soon as possible) and ALAP (as late as possible) scheduling algorithms[10]. Two tasks can be combined only if their time frames overlap.

Specifically, the ASAP algorithm assigns time 1 to the task without predecessor in a conditional block, i.e, the fork task, and then, time 2 to its immediate successors and so on. The ALAP algorithm first assigns time e to the tasks whose immediate successor is the join task of the conditional block, then, time $e-1$ to their immediate predecessors and so on. For example, when $e=7$, the time frames of tasks in Fig. 1 are shown in **Fig. 4.**

We now describe several algorithms to choose the tasks to be combined. The algorithms can be divided into two classes. One class contains algorithms that only utilize the goodness of combining two tasks each time a choice is made and disregard the effect of the combination to other combinations. We call algorithms in this class algorithms without total goodness estimation. The other class contains algorithms that attempt to estimate the total goodness at the end of the transformation.

4.1 Algorithms without Total Goodness Estimation

There are four algorithms in the class of algorithms without total goodness estimation. The first one is a straightforward and the simplest algorithm in terms of complexity. This algorithm is called *Fork Random* algorithm and is summarized by the pseudo code below. The algorithm randomly chooses one node with *level* 2 and find all combinations that include the chosen node. (The level of a node is the shortest path length to the node from the fork node. The level of the immediate successors of the fork node is 2.) One combination is chosen randomly and the nodes are combined. After this first iteration, pairs are similarly found for the rest of nodes with level 2. The algorithm then proceeds to the next higher level nodes. Time complexity of this algorithm is $O(n^2)$ where n is the number of tasks.

```
Input: The given precedence graph $G=(V, E)$.
Output: The transformed precedence graph
        without conditional branches.
Algorithm: (Fork Random)
  Compute $S$={all nodes at level 2};
  $i=2$;
  WHILE $i$ is less than the longest path
  length of the conditional block DO
    WHILE $S$ is not empty DO
      SET $Q$ empty;
      Randomly choose $\tau_j$ from $S$;
```

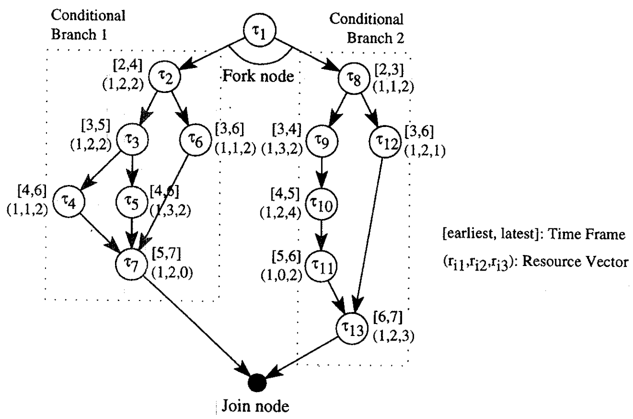


Fig. 4 Time frames.

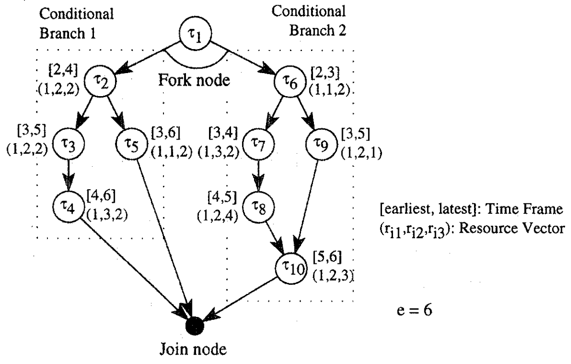


Fig. 5 Example task set.

```
            Find all $\tau_k$s that are not in the same
              conditional branch as $\tau_j$ and whose
              time frames overlap with the time
              frame of $\tau_j$ ;
            Add all combination $(\tau_j, \tau_k)$s into $Q$ ;
            Randomly choose a combination $(\tau_x,
              \tau_y)$ from $Q$, combine nodes $\tau_x$ and $\tau_y$,
              and replace them with a new node
              $\tau_{(x,y)}$ ;
            Remove $\tau_x$ and $\tau_y$ from $S$ if they are in
              $S$ ;
        END
        $i \leftarrow i+1$ ;
        Compute $S$={all nodes at level $i$} ;
    END
```

In the example in **Fig. 5**, S is set to $\{\tau_2, \tau_6\}$ initially. Time frames are computed assuming $e=6$. Suppose τ_2 is chosen. Q includes (τ_2, τ_6), (τ_2, τ_7), (τ_2, τ_8), and (τ_2, τ_9). The algorithm randomly chooses one combination from Q and combine the pair of nodes in it. The algorithm may choose (τ_2, τ_7) randomly. τ_2 is removed from S and now $S=\{\tau_6\}$.

The second algorithm is similar to the first one. As the previous algorithm, one node that has level 2 is randomly chosen. The algorithm finds all possible combinations that include the node. Instead of choosing randomly from the possible combinations, a combination (τ_x, τ_y) with the maximum goodness of combination $R_{(x, y)}$ is chosen. Ties are broken by combining the nodes with the minimum difference of levels. After this first iteration, we repeat the above until combinations are found for all immediate successors of the fork node. The algorithm then proceeds to the next higher level nodes. We call this algorithm *Fork First* algorithm. The algorithm is summarized by a pseudo code below. Time complexity of this algorithm is $O(mn^2)$ where m is the number of resource types and n is the number of tasks.

```
  Input : The given precedence graph $G=(V,
          E)$.
  Output : The transformed precedence graph
           without conditional branches.
  Algorithm : (Fork First)
    Compute $S$={all nodes at level 2} ;
    $i=2$ ;
    WHILE $i$ is less than the longest path
        length of the conditional block DO
      WHILE $S$ is not empty DO
        SET $Q$ empty ;
        Randomly choose $\tau_j$ from $S$ ;
        Find all $\tau_k$'s that are not in the same
          conditional branch as $\tau_j$ and whose
          time frames overlap with the time
          frame of $\tau_j$ ;
        Add all combination $(\tau_j, \tau_k)$s into $Q$ ;
        Compute $R_{(x,y)}$ for all combination $(\tau_x,
          \tau_y)$s in $Q$ ;
        Choose the combination $(\tau_x, \tau_y)$ with
          the maximum $R_{(x,y)}$ from $Q$, com-
          bine nodes $\tau_x$ and $\tau_y$, and replace
          them with a new node $\tau_{(x,y)}$ ;
        (If there are ties, a combination with
          the minimum difference between
          levels of $\tau_x$ and $\tau_y$ is chosen.)
        Remove $\tau_x$ and $\tau_y$ from $S$ if they are in
          $S$ ;
      END
      $i \leftarrow i+1$ ;
      Compute $S$={all nodes at level $i$} ;
    END
```

In the example in Fig. 5, $S=\{\tau_2, \tau_6\}$ for $i=2$. Suppose τ_2 is chosen. Q includes (τ_2, τ_6), (τ_2, τ_7), (τ_2, τ_8), and (τ_2, τ_9). When all α_is are 1 in the definition $R_{(x,y)}$, the combinations (τ_2, τ_7) and (τ_2, τ_8) have the maximum $R_{(x,y)}=6$ among the combinations in Q. Since the levels of τ_2, τ_7, and τ_8 are 2, 3, and 4 respectively, we combine (τ_2, τ_7). The final result by this algorithm may be as shown in **Fig. 6**(a). The resource sharing index R of this transformation is 18.

The third algorithm does not start by finding combinations for nodes in level 2. Instead, it computes $R_{(x,y)}$ for all possible combinations and chooses the one with the largest $R_{(x,y)}$. We call this algorithm *Best First* algorithm. Time complexity of this algorithm is $O(mn^3)$. This algorithm is the most expensive algorithm in terms of complexity among the four algorithms in this class.

```
Input: The given precedence graph $G=(V, E)$.
Output: The transformed precedence graph without conditional branches.
Algorithm: (Best First)
  SET $S=\{(\tau_i, \tau_j) \mid$ time frames of $\tau_i$ and $\tau_j$ overlap and $\tau_i$ and $\tau_j$ are not in the same conditional branch$\}$;
  Compute $R_{(i,j)}$ for all $(\tau_i, \tau_j)$ in $S$;
  WHILE $S$ is not empty DO
    Combine nodes $\tau_i$ and $\tau_j$ with the maximum $R_{(i,j)}$;
    (If there are ties, choose the combination with the minimum difference of levels)
    SET $S=\{(\tau_i, \tau_j) \mid$ time frames of $\tau_i$ and $\tau_j$ overlap and $\tau_i$ and $\tau_j$ are not in the same conditional branch$\}$;
    Compute $R_{(i,j)}$ for all $(\tau_i, \tau_j)$ in $S$;
  END
```

In the example in Fig. 5, S is set to $\{(\tau_2, \tau_6), (\tau_2, \tau_7), (\tau_2, \tau_8), (\tau_2, \tau_9), (\tau_3, \tau_6), (\tau_3, \tau_7), (\tau_3, \tau_8), (\tau_3, \tau_9), (\tau_3, \tau_{10}), (\tau_4, \tau_7), (\tau_4, \tau_8), (\tau_4, \tau_9), (\tau_4, \tau_{10}), (\tau_5, \tau_6), (\tau_5, \tau_7), (\tau_5, \tau_8), (\tau_5, \tau_9), (\tau_5, \tau_{10})\}$. After computing $R_{(i,j)}$ for all $(\tau_i, \tau_j) \in S$, we find $R_{(4,7)}=6$ is the maximum. We combine τ_4 and τ_7. This step is repeated until possible combinations are exhausted. The final result by this algorithm is shown in Fig. 6 (b). The resource sharing index R of this transformation is 18.

The fourth algorithm is an approximation of the third algorithm. Instead of finding the maximum $R_{(i,j)}$, the algorithm finds a combination for the task with the maximum resource requirement first. In this algorithm, we sort task τ_is according to their resource requirements. The resource requirement of τ_i is

$$R_i = a_1 r_{i1} + \cdots + a_m r_{im},$$

where a_is are the same as in the definition of $R_{(x,y)}$. The algorithm starts by finding all possible combinations for the task τ_x that has the largest resource requirement R_x. We choose combination (τ_x, τ_y) with the largest $R_{(x,y)}$. We call this algorithm *Largest First* algorithm. Time complexity of this algorithm is $O(mn^2)$.

```
Input: The given precedence graph $G=(V, E)$.
Output: The transformed precedence graph without conditional branches.
Algorithm: (Largest First)
  Construct a sorted list $L$ of tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$ such that $R_1 \geq R_2 \geq \cdots \geq R_n$;
  WHILE $L$ is not empty DO
    Remove a task from the front of $L$ and name the task $\tau_i$;
    Set $Q=\{\tau_j \mid \tau_j$'s time frame overlaps with the time frame of $\tau_i$ and $\tau_i$ and $\tau_j$ are not in the same conditional branch$\}$;
    Compute $R_{(i,j)}$ for all $\tau_j$ in $Q$;
    Combine nodes $\tau_i$ and $\tau_j$ with the maximum $R_{(i,j)}$;
    (If there are ties, choose the combination with the minimum difference of levels)
    Remove $\tau_j$ from $L$;
  END
```

Using the example in Fig. 5, we note that L is initially set to $\{\tau_8, \tau_4, \tau_7, \tau_{10}, \tau_2, \tau_3, \tau_5, \tau_6, \tau_9\}$. The first τ_i is τ_8 and Q is set to $\{\tau_2, \tau_3, \tau_4, \tau_5\}$. $R_{(8,2)}=5$, $R_{(8,3)}=5$, $R_{(8,4)}=5$, and $R_{(8,5)}=4$. Since differences in levels between τ_8 and τ_2, between τ_8 and τ_3, and between τ_8 and τ_4 are 2, 1, and 0 respectively, we choose τ_8 and τ_4 as the first combination. τ_4 is removed from L and now $L=\{\tau_7, \tau_{10}, \tau_2, \tau_3, \tau_5, \tau_6, \tau_9\}$. This step is repeated for the first task in L. The final result by this algorithm is shown in Fig. 6 (c). The resource sharing index R of this transformation is 17.

4.2 Algorithms with Total Goodness Estimation

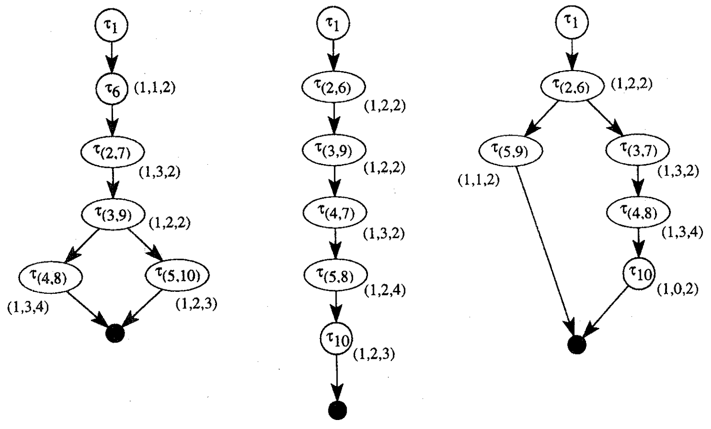
In this class of algorithms, we select the most promising combination by estimating the value of the resource sharing index R if the combination were indeed selected. In contrast, only the goodness $R_{(x,y)}$ of a combination is used as the selection criteria in the Fork First, Best First, and Largest First algorithms.

The estimation of R when tasks τ_i and τ_j are combined is denoted by $\hat{R}(\tau_i, \tau_j)$. We use the probability that a task will be scheduled at a particular time within its time frame to compute this estimate. The probability is derived by the same way as the force-directed scheduling algorithm[10] does. The force-directed scheduling algorithm assumes that the task is equally likely to be scheduled at any time t in its time frame. Hence, the probability for a task τ_i with a time frame of length k to be scheduled at any time t in its time frame is $1/k$. For example, suppose a task τ_1 has a time frame [2, 5], then τ_1 may be scheduled at time 2, 3, 4, and 5, with each probability 0.25.

The estimation of R is computed according to

$$\hat{R}(\tau_i, \tau_j) = a_1 \sum_{t=1}^{e} \min_{\forall x}(A_1^x(t)) + a_2 \sum_{t=1}^{e} \min_{\forall x}(A_2^x(t)) + \cdots + a_m \sum_{t=1}^{e} \min_{\forall x}(A_m^x(t)).$$

In this equation $A_l^x(t) = \sum_{\tau_k \in \Omega_x}(r_{kl}\pi_k(t))$, where



(a) Fork First (b) Best First (c) Largest First

Fig. 6 Transformation results.

$\Omega_x=\{\tau_i \mid \tau_i$ is in x-th conditional branch$\}$ and $\pi_k(t)$ is the probability with which task τ_k is scheduled at time t if τ_i and τ_j are combined. $r_{kl}\pi_k(t)$ is the expected number of resource ρ_l used by τ_k at time t. Therefore, $A_l^x(t)$ is the expected number of resource ρ_l used at time t if x-th conditional branch is taken. By taking the minimum of $A_l^x(t)$ over all branches, we find the number of resource ρ_l utilized at time t whichever branch is taken. By maximizing $\hat{R}(\tau_i, \tau_j)$, we are maximizing the expected number of conditionally shared resources.

It is also possible to have the same variety of algorithms in this class as in the class where algorithms do not estimate the total goodness:

- combining from the fork node (Fork First with Est),
- combining the best match first (Best First with Est), and
- combining the largest resource requirement first (Largest First with Est).

We use the names inside the parentheses to refer to the algorithms. The pseudo code describing each of the algorithms with the total goodness estimation is the same as that for the corresponding algorithm without the estimation, which is given in the previous section, the only difference being that $\hat{R}(\tau_i, \tau_j)$ is used instead of $R_{(i,j)}$. Time complexity of computing $\hat{R}(\tau_i, \tau_j)$ is $O(emn)$ while complexity of computing $R_{(i,j)}$ is $O(m)$. Therefore, time complexities of Fork First with Est, Best First with Est, and Largest First with Est are $O(emn^3)$, $O(emn^4)$, and $O(emn^3)$, respectively.

4.3 Schedule Generation

Given a precedence graph with OR constraints, we transform the graph into one without OR constraints using one of the algorithms explained in Sections 4.1 and 4.2. Then, the tasks in the transformed precedence graph are scheduled using some classic scheduling algorithms[2),9)] that are applicable to task sets with precedence relations and resource requirements, and without OR constraints. Finally, from the schedule of the union tasks, we extract the schedule of the original tasks in order to execute tasks.

Suppose the task set in Fig. 5 is given. We assume the system is a single processor system here in order to depict the process of extracting the schedule of the original tasks from the schedule of the union tasks. Using the Largest First algorithm, we transform the precedence graph into the one in Fig. 6 (c). Using the HLFET algorithm,[1)] we may schedule the transformed task set as schedule (a) in **Fig. 7**. From this schedule, we can extract either schedule (b) or (c) in this figure. Schedule (b) is used if the conditional branch 1 in Fig. 5 is taken and schedule (c) is used if the conditional branch 2 is taken. These schedules can be easily extracted from schedule (a) by picking up tasks in the corresponding conditional branch. Time 6 in schedule (b) is idle, i.e., if the conditional branch 1 in Fig. 5 is taken, we must keep the processor idle at time 6.

5. Simulation

To evaluate the algorithms, we performed a set of simulations. When there are many conditional blocks in a precedence graph, the innermost conditional block is transformed first and

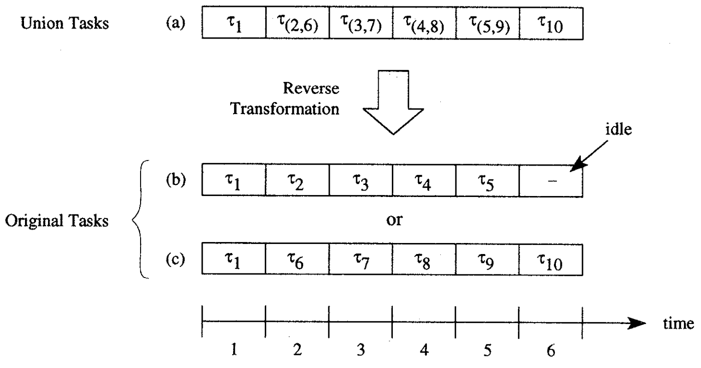


Fig. 7 Schedule.

Table 1 Task generation parameters.

$L_{longest}$	Longest path length
$N_{resource\_types}$	The number of resource types
$minN_{Ri}$	The minimum number of resource type-i
$maxN_{Ri}$	The maximum number of resource type-i
$minN_{branch}$	The minimum number of AND branches in each conditional branch
$maxN_{branch}$	The maximum number of AND branches in each conditional branch
$avgL_{branch}$	Average AND branch length
ΔL_{branch}	Variance of AND branch length
$avgP_{right\_branch}$	Average probability to shorten the longest path in the right branch

the transformation proceeds to outer conditional blocks. Therefore, we only need to compare performance on a single conditional block to compare transformation algorithms.

5.1 Task Set Generation

We compare the transformation algorithms on synthesized task sets with two conditional branches. Each task set is generated according to the following algorithm:

```
Input: Parameters in Table 1.
Output: A task set with two conditional
        branches.
Algorithm: (Task Set Generation)
  Read the longest path length of the conditional block from input and assign the path length to the left conditional branch;
  Generate resource requirements of nodes on the longest path;
  Generate the number of AND branches $N_L$ in the left conditional branch;
  FOR $i=1$ TO $N_L$ DO
    Generate branch length and resource requirements of nodes on the branch;
    Decide a node to branch out;
    Decide a node to merge in;
    (A branch merges on a node in the branch where the branch has branched out.)
  END
  Generate the longest path length of the right conditional branch;
  Generate resource requirements of nodes on the longest path in the right conditional branch;
  Generate the number of AND branches $N_R$ in the right conditional branch;
  Generate $N_R$ branches for the right conditional branch.
```

The task set generation parameters are summarized in Table 1. $L_{longest}$ is the longest path length of the conditional block to be generated. $L_{longest}$ is assigned to the left conditional branch. We use 10 and 20 as $L_{longest}$ in our simulation. These values are chosen arbitrary but two values are chosen to compare the effect of different $L_{longest}$. $N_{resource\_types}$ is the number of resource types in the system. In the simulation, $N_{resource\_types}=3$. Resource requirements of nodes are generated using a uniform random number generator with its minimum and maximum numbers given as parameters. $minN_{Ri}$ and $maxN_{Ri}$ are the minimum and the maximum number of resource type i to be used by a generated task. Uniform distribution is chosen to reflect variety of resource requirements of tasks. The number of AND branches in each conditional branch is generated by a uniform random number generator with given maximum and minimum value. $minN_{branch}$ and $maxN_{branch}$ are the minimum and the maximum number of AND branches in each conditional branch to be generated. $minN_{branch}$ is set to 1 and $maxN_{branch}$ is changed. An *AND branch* is a sequence of tasks with straight line precedence constraints. The length of an AND branch is generated by a truncated normal random number generator in order to control the mean and variance separately. $avgL_{branch}$ and ΔL_{branch} are an average and a variance of AND branch lengths. ΔL_{branch} is set to 1 and $avgL_{branch}$ is varied. The nodes where AND branches go out and merge in are chosen randomly from nodes which do not lengthen the longest path. The longest path length of the right branch is generated by a binomial random number generator. We chose binomial distribution so that the left branch always has the longest path and the longest path length of the right branch is correlated to the longest path length of the graph. $avgP_{right\_branch}$ is the average probability to shorten the longest path in the right branch. In the simulation, the right branch is one node shorter than the left branch with 5% chance.

5.2 Simulation Results

Since the effectiveness index of a transformation, R/e, may be very different from task set to task set, we compare difference of the value R/e in order to compare performance of the algorithms instead of R/e itself. We use Fork Random algorithm described in Section 4.1 as the basis of the comparison. The value we compare is defined by

$$G=\frac{R}{e}-\frac{(R \text{ produced by Fork Random})}{e}.$$

The simulation result shown in **Fig. 8** is produced with the parameter set in **Table 2**. The values shown in the figure are averages. The 95% confidence intervals are less than 0.05 on each side of the plotted values.

The value R/e by Fork Random with the same parameters is shown in **Fig. 9**. The intervals shown above and below the plots are 95% confidence intervals. The decrease of R/e in Fig. 9 is more than the increase of G in Fig. 8. Therefore, the maximum R/e is likely to be achieved with the minimum e for all the algorithms. In this sense, the value e to be used is not very important. We only need a little larger value e than the longest path length of the conditional block.

Among all the algorithms, Best First that does not use the total goodness estimation performs best. The second best algorithm, Largest First, also does not use the estimation. If we compare the two classes of algorithms, with and without the estimation, each algorithm without the estimation works better than the respective counterpart with the estimation. Without the estimation, algorithms seek the most gain on the value R in each step of transformation. On the other hand, algorithms with the estimation ignore immediate gain while they try to maximize R at the end. The result shows that we can expect better return by being greedy. This result is preferable because we can save computation by skipping the estimation.

Among algorithms without the estimation, Best First works best as expected. On the other

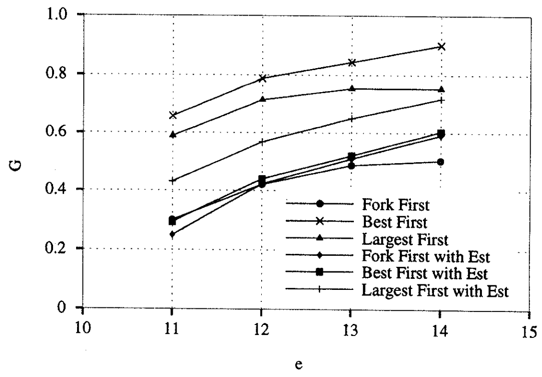


Fig. 8 Simulation result 1.

Table 2 Simulation parameters.

$L_{longest}$	10
$N_{resource\_types}$	3
$minN_{R1}$	1
$maxN_{R1}$	2
$minN_{R2}$	0
$maxN_{R2}$	3
$minN_{R3}$	0
$maxN_{R3}$	4
$minN_{branch}$	1
$maxN_{branch}$	4
$avgL_{branch}$	2
ΔL_{branch}	1
$avgP_{right\_branch}$	0.95

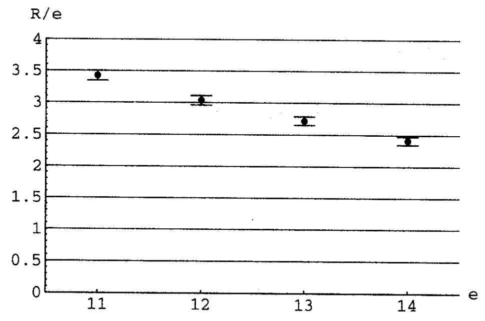


Fig. 9 R/e by fork random.

hand, Largest First with Est works better than Best First with Est. Largest First with Est not only looks for good final R but also considers immediate gain by finding a combination for the task with the maximum resource requirements. Since seeking immediate gain is preferable for a good performance, this property of Largest First with Est provides better performance than Best First with Est.

Considering the confidence interval, we can not find any significant performance difference between Fork First with Est and Best First with Est algorithms. Also, their performances are among the worst in the six algorithms. This result suggests that the total goodness estimation we used is not a very good estimation. The estimation can be improved by using accurate $\pi_k(t)$, the probability with which task τ_k is scheduled at time t. However, improving the estimation increases complexity. Since the transformation is the only one of three stages in our scheduling process, spending much time on this stage is not a good choice in our opinion.

Figure 10 shows the average longest path length after the transformation for given e. The algorithms without the estimation produce almost the same longest path length as given e. On the other hand, the algorithms with the estimation produce shorter longest path length than given e as e increases. When $\tilde{R}(\tau_i, \tau_j)$ is computed, the expected number of conditionally shared resources at each time t, $1 \le t \le e$, i.e., $\min_{\forall x}(A_i^x(t))$, is summed up. Lengthening the longest path means that some nodes are forced to have no combination candidates. Contribution of such a task to $\tilde{R}(\tau_i, \tau_j)$ is zero unless other union tasks happens to be executed with it at the same time. Also, lengthening the longest path lengthens time frames, and consequently decreases $\pi_k(t)$. These two factors reduce $\tilde{R}(\tau_i, \tau_j)$ when the longest path is lengthened. Since all possible combinations are considered by Best First with Est algorithm, the algorithm eliminates the combinations that lengthen the longest path. With Fork First with Est and Largest First with Est algorithms, the chance to eliminate such combinations is less because these algorithms consider a limited set of combinations at each step of transformation process. When a combination for a node is searched in Fork First with Est algorithm, all nodes at lower levels than the node in question have already found their combinations. On the other hand, some nodes at lower levels may not have found their combinations when a combination for a node is searched in Largest First with Est algorithm. Therefore, it is more likely for the longest path to be lengthened in the former case than in the latter case. For this reason, Fork First with Est has tendency to lengthen the longest path.

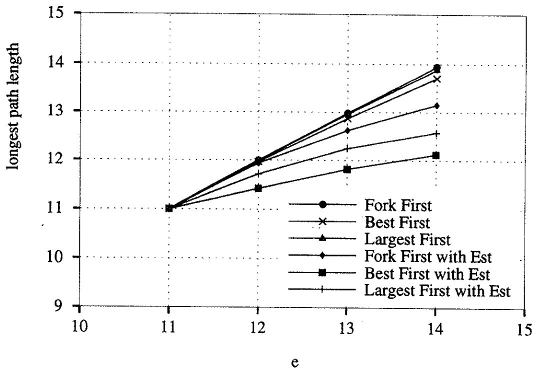


Fig. 10 Average longest path length.

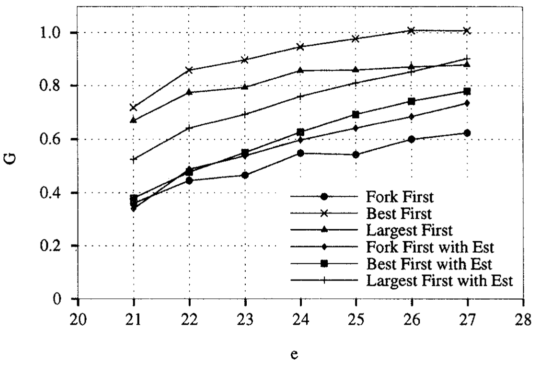


Fig. 11 Simulation result 2 (Longer path length).

Next, we lengthen the longest path length $L_{longest}$ to 20 and kept other parameters, except $avgL_{branch}$, at their values in Table 2. The value G becomes as in Fig. 11. The 95% confidence intervals are less than 0.05 on each side of the plotted values. We change $avgL_{branch}$ to 4 so that the average length of AND branches has the same ratio with the longest path length as in the case of Fig. 8. **Figure 11** has the similar characteristics to Fig. 8.

As e increases, G of an algorithm with the estimation increases more than G of an algorithm without the estimation in Fig. 11. This characteristics present in Fig. 8 also. The larger the value e is, the more possible combinations to consider at each step of the transformation

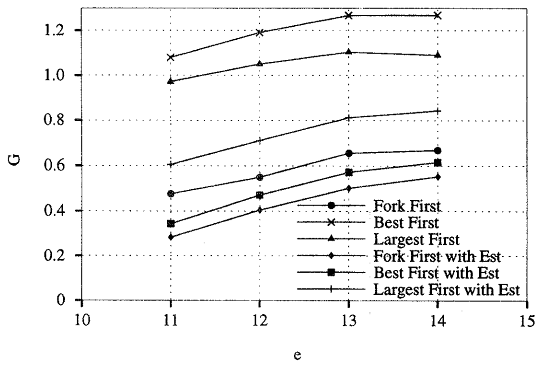


Fig. 12 Simulation result 3 (More branches).

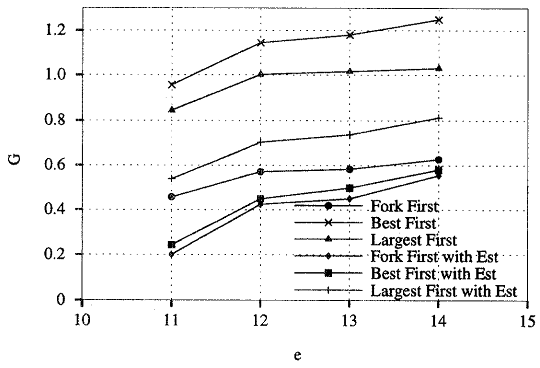


Fig. 13 Simulation result 4 (Longer branches).

process exist. Choosing a certain combination may eliminate good combinations that the algorithm may encounter later. Algorithms without the estimation perform more combinations that eliminate good future combinations than algorithms with the estimation. Since large e provides many possible combinations, the effect of poor choices of the algorithms without the estimation at early stages becomes significant. For this reason, the performance of algorithms with the estimation improves more than algorithms without the estimation as e increases.

When the number of AND branches is increased, i.e., $maxN_{branch}=8$, or AND branches are lengthened, i.e., $avgL_{branch}=4$, the value G becomes as shown in **Fig. 12** or **Fig. 13**. Other parameters are the same as Table 2. The 95% confidence intervals are less than 0.05 on each side of the plotted values. The difference between the algorithms with and without the estimation becomes clearer than in Fig. 8. With more AND branches or longer AND branches, there are more possible combinations as with larger e. However, in contrast to the case where e is large, lengthening the longest path is not allowed in this case. Consequently, with many or long AND branches, the algorithms without the estimation are less likely to choose a combination that makes some nodes impossible to find a combination than with large e. Also, by having more possible combinations, the algorithms without the estimation have more chance to find larger $R_{(i,j)}$ at each step of the transformation process. Increasing the number of AND branches and lengthening AND branches create a situation in which the algorithms without total goodness estimation work well.

In summary, Best First works best and Largest First is the second best. The performance of Largest First with Est improves as e increases. Fork First with Est and Best First with Est work poorly in all cases. Besides all algorithm mentioned above, Fork Random algorithm works worst. However, Best First algorithm is expensive in terms of complexity. Computation time of Best First algorithm increases in the third power of the number of tasks while computation time of Largest First algorithm increases quadratically with the number of tasks. From this observation, Best First algorithm may not be the best choice when the task set is large. Largest First algorithm is effective with less computation.

6. Conclusion

In this paper, we propose several strategies to transfer a task precedence graph with conditional execution to a graph without. The transformation simplifies deterministic scheduling of tasks with conditional execution and resource sharing. The scheduling is performed in three stages. In the first stage, conditional execution is eliminated from the precedence relation. Then, the transformed task set, which does not have conditional execution anymore, is scheduled using a classical scheduling algorithm that is applicable to a task set without conditional execution. Finally, the schedule for the transformed task set is projected to the schedule for the original task set. This paper describes detail of the first stage.

We have found that greedy algorithms work better than algorithms that attempt to maximize expected final results. The best algorithm, Best First algorithm, is somewhat expensive in terms of complexity. However, there is an alternative algorithm, Largest First algorithm, which is less expensive than Best First algorithm and produce fairly good results.

Several issues need to be investigated in future research. In our algorithms, nested conditional blocks are handled from innermost block and we only need to look at a single conditional block at a time. This approach reduces complexity of the algorithms and is an advantage of our algorithms. However, if we can incorporate some global view and utilize the global information, performance of the transformation may be improved in case of nested conditional blocks.

We have not considered the scheduling algorithm and assumed some scheduling algorithm is used to schedule the transformed task set. However, we may be able to schedule the original task set more effectively by transforming it to a task set with a certain property or there may be a scheduling algorithm better suited for scheduling the transformed task set. Studying the coupling between the transformation stage and the scheduling stage is another direction for the extension of our work.

Acknowledgement This research was supported in part by the Office of Naval Research under grant N00014-92-J-1146 and N00014-89-J-1181.

References

1) Adam, T. L., Chandy, K. M. and Dickson, J. R.: A Comparison of List Schedules for Parallel Processing Systems, *Comm. ACM*, Vol. 17, No. 12, pp. 685-690 (1974).

2) Blazewicz, J.: Selected Topics in Scheduling Theory, *Annals of Discrete Mathematics*, No. 31, pp. 1-60 (1987).

3) Camposano, R.: Path-Based Scheduling for Synthesis, *IEEE Trans. on Computer Aided Design*, Vol. CAD-10, No. 1, pp. 85-93 (1991).

4) Garey, M. R. and Johnson, D. S.: Complexity Results for Multiprocessor Scheduling under Resource Constraints, *SIAM Journal of Computing*, Vol. 4, No. 4, pp. 397-411 (1975).

5) Garey, M. R. and Johnson, D. S.: *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York (1979).

6) Gillies, D. W. and Liu, J. W. S.: Scheduling Tasks with and/or Precedence Constraints, Technical Report UIUCDCS-R-91-1627, Dept. of Computer Science, University of Illinois at Urbana-Champaign, IL (Mar. 1991).

7) Kim, T., Liu, J. W. S. and Liu, C. L.: A Scheduling Algorithms for Conditional Resource Sharing, *Proc. International Conf. on Computer-Aided Design*, pp. 84-87 (1991).

8) Lawler, E. L.: *Combinatorial Optimization : Networks and Matroids*, Holt, Rinehart and Winston, New York (1976).

9) Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. and Shmoys, D. B.: Sequencing and Scheduling: Algorithms and Complexity, Technical Report BS-R8909, Centre for Mathematics and Computer Science, Amsterdam (June 1989).

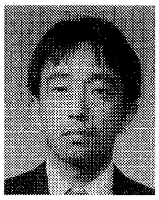
10) Paulin, P. G. and Knight, J. P.: Force-Directed Scheduling for the Behavioral Synthesis of Asic's, *IEEE Trans. on Computer-Aided Design*, Vol. CAD-8, No. 6, pp. 661-679 (June 1989).

11) Tseng, C.-J., Wei, R. W., Rothweiler, S. G., Tong, M. and Bose, A. K.: Bridge: A Versatile Behavioral Synthesis System, *Proc. 25th Design Automation Conference*, pp. 415-420 (1988).

12) Ullman, J. D.: NP-Complete Scheduling Problems, *Journal of Computer and System Sciences*, Vol. 10, No. 3, pp. 384-393 (1975).

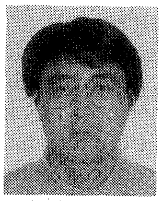
13) Wakabayashi, K. and Yoshimura, T.: A Resource Sharing Control Synthesis Method for Conditional Branches, *Proc. International Conf. on Computer-Aided Design*, pp. 62-65 (1989).

(Received November 4, 1994)
(Accepted May 12, 1995)



Hidenori Nakazato received his B. Engineering degree in electronics and telecommunications from Waseda University in 1982 and his MS and PhD degrees in computer science from University of Illinois in 1989 and 1993, respectively. He is currently a Research Manager at Oki Electric, Japan. His research interests include real-time systems, distributed systems, and databases. He is a member of IPSJ.

Jane W. S. Liu received her Bachelor of Science degree in Electrical Engineering from the Cleveland State University, Ohio. She received her Master of Science and Electrical Engineers degrees and her Doctor of Science degree from the Massachusetts Institute of Technology. She is currently a Professor of Computer Science at the University of Illinois at Urbana-Champaign. Her research interests are in the areas of real-time systems, distributed systems and computer networks. Before joining the University of Illinois in 1973, she worked as an electronics engineer for the U. S. Department of Transportation, Transportation Systems Center, Cambridge Massachusetts, as a Member of the Technical Staff of the Mitre Corporation, Bedford, Massachusetts, and as an engineer for the Radio Corporation of America, Needham, Mass. She is an IEEE Fellow and a member of ACM.



Taewhan Kim received the B. S. degree in computer science and statistics and the M. S. degree in computer science from Seoul National University, Seoul, Korea in 1985, 1987, respectively, and received the Ph. D. degree in computer science from the University of Illinois at Urbana-Champaign in 1993. He is currently a Senior CAD Software Engineer at Lattice Semiconductor Corporation, USA. His research interests include high-level and logic synthesis for VLSI.