An Optimal Instruction Scheduler for Superscalar Processor

Hong-Chich Chou and Chung-Ping Chung

***Abstract*—Performance in superscalar processing strongly depends on the compiler's ability to generate codes that can be executed by hardware in an optimal or near optimal order. Generating optimal code is an NP-complete problem. However, there is a need for highly optimized code, such as in superscalar or real-time systems. In this paper, an instruction scheduling scheme for optimizing a program trace is proposed. Optimized code can be arrived at without much redundant work, if some important features in code are well explored and utilized in scheduling. To formalize the task, two abstract models, one for a superscalar processor and the other for a program trace, are given. These two models reflect most of the characteristics of the scheduling problem. The interrelations between instructions and partial schedules are thoroughly studied, and dominance and equivalence relations on them are defined. These relations are then used to reduce the solution space and eventually help to produce optimal schedules. The results of experiments that show the promise of the proposed scheme are also presented.**

***Index Terms*—Pipeline processors, sequencing and scheduling, optimization, prune and search, and NP-completeness.**

I. Introduction

SUPERSCALAR processors, in contrast to scalar or vector processors, are processors which attempt to improve the execution rate of programs by executing, on average, more than one instruction per clock cycle. Examples of superscalar processors include the IBM RS/6000, Intel i860 [1], [2], and DEC Alpha. The success of superscalar machines depends not only on the vast hardware resources they provide, but even more importantly, on how efficiently these resources are used. The objective of superscalar instruction scheduling is to rearrange instructions in such a way that they are executed by hardware in an optimal (or near optimal) order (i.e., so that execution time is minimized).

In this paper, an optimization scheduling scheme specially tailored for superscalar processing is proposed, since no satisfactory scheme exists at present. Though the instruction scheduling problem appears similar to the unit-execution-time (UET) task scheduling problem, there are some fundamental differences between them. Superscalar instructions are typed and may induce delay cycles in terms of result availability. These two features are not seen in UET task scheduling. Another type of delay is in task scheduling—the communication delay [3]. Furthermore, pipelining is not possible at the task level, although it is a must at the instruction level. There are several reasons for studying superscalar processing optimization schemes: 1) there is a need for highly optimized code, such as in real time applications, 2) an optimized code can be used to evaluate a heuristic schedule, and 3) scheduling principles can be found in the research, which can guide the design of superscalar architecture and heuristic algorithms.

Instruction scheduling can be done on intermediate code or object code or both [4], [5]. From the instruction scheduling point of view, intermediate code is different from object code in that object code may contain, in addition to data dependencies, output and anti-dependencies or restrictions imposed by hardware. Thus, scheduling intermediate code is regarded as a subproblem of scheduling object code. So, in this paper we consider only the scheduling of object codes.

Instruction scheduling can be done by hardware or by software [6], [7]. However, to be fully functional, hardware scheduling must be able to solve several problems, including artificial dependencies caused by register-file limitations, conditional branches, imprecise interrupts, and so forth. Hardware capable of solving all these problems is complex and costly, and, more seriously, is slow in clock rate.

In contrast to hardware scheduling, the task of identifying parallel executable instructions and scheduling them can be done by software. The VLIW machine [11] is an extreme example, which relies completely on software instruction scheduling to keep functional units busy while avoiding data hazards [8]. Except for identifying parallel executable instructions, the scheduling software also tries to optimize *delay slots*. Since software exposes program parallelism to processors, the processor can be much simplified by assuming that the incoming instructions in each cycle can be executed simultaneously, as in the approach used in the IBM RS/6000, or by setting a bit in an instruction to notify the processor to execute the next instructions in parallel, as in the approach used in the Intel i860. A simpler hardware design often implies a faster clock rate.

Generally, the optimization superscalar instruction scheduling problem is intractable [9], [10]. Our optimization scheme focuses on reducing solution space and execution time as much and as early as possible. Nevertheless, we believe that an optimal schedule can be arrived at without much redundant work, if some important relations among instructions and among partial schedules can be thoroughly explored and utilized.

Manuscript received December 21, 1992; revised October 2, 1993. This work was supported in part by the National Science Council, Republic of China, under Contract NSC 81-0408-E-009-594.

The authors are with the Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

IEEE Log Number 9408141.

II. Formal Specifications

In this research, we begin by conceiving two abstract models, one for a superscalar architecture and another for the program trace to be scheduled. These two models reflect most of the flavor of superscalar instruction scheduling problem and, formalize our research tasks.

A superscalar processor can be represented by a 4-tuple notation $(k, M, \mathcal{P}, z)$, where

- $k \geq 1$ denotes the number of different types of processors employed in the superscalar architecture.
- $M = (m_1, m_2, \cdots, m_k)$, where m_i denotes the number of type i processors, for $1 \leq i \leq k$. The instruction set is also categorized into k types and, type i instructions can only be executed on any of the m_i type i processors.
- $\mathcal{P} = \{P_{ij} | 1 \leq i \leq k, 1 \leq j \leq m_i\}$ denotes the set of processors. P_{ij} denotes the jth processor of type i.
- $z \geq 0$ denotes the maximum number of pipeline delay cycles required to execute any instruction.

The above machine model is generalized enough to encompass a wide range of superscalar architecture. For example, for the IBM RS/6000, $k = 4$, $m_1 = m_2 = m_3 = m_4 = 1$ and $z = 6$ (delay of floating-point compare instruction).

We formally represent a program trace by a 4-tuple notation $(U, \prec, Pf, D)$, where

- $U = \{I_1, I_2, \cdots, I_i\}$ is the set of instructions constituting a program trace.
- $\prec$ is a transitive binary relation defined on U that specifies the precedence relationships among instructions.
- Pf is a function, $Pf: U \rightarrow \{1, 2, \cdots, k\}$, that specifies the type of processor on which an instruction is to be executed.
- D is a function, $D: U \rightarrow \{0, 1, \cdots, z\}$, that specifies the number of delay cycles of an instruction.

Here a program trace in our concern denotes a segment of code which are to be scheduled in a batch. If the segment contains only a single basic block, the scheduling may be called local; otherwise it is global [12]. Strategies for increasing instruction parallelism, such as loop unrolling or basic block enlargement [13], are beyond the scope of this paper.

The three main restrictions which limit instructions from being executed in parallel, namely, 1) data dependencies, 2) procedural dependencies, and 3) resources conflicts [7], are reflected in these models. Precedence relations $\prec$ originate from true data dependencies, procedural dependencies, antidependencies, and output dependencies. If $I_i \prec I_j$, then I_i must be executed before I_j. The terms *dependence* and *precedence* are used interchangeably below.

Resource conflicts are reflected in the function $Pf()$. We assume that there is no resource conflict between instructions of different types. And the only resource conflict between instructions of the same type is the contention for processors when instructions to be issued are more than the number of processors available. This assumption is feasible for superscalar instructions.

It will be convenient to represent each instance $(U, \prec, Pf, D)$ by a directed acyclic graph (DAG). Fig. 1, for example, shows a DAG with three types and a maximum delay of two cycles. Nodes in Fig. 1 denote instructions. If $I_i \prec I_j$, then there is a directed edge from node I_i to node I_j in the graph (note that transitive edges are not shown). The number after the slash in each node indicates the type of that instruction, and the number beside each edge denotes the number of delay cycles required by the predecessor. Fig. 1 will be used as a running example throughout this paper. Conventional terminology for a graph is followed here. If $I_i \prec I_j$, then $I_j(I_i)$ is a successor (predecessor) of $I_i(I_j)$. If there is no I' such that $I_i \prec I' \prec I_j$, then $I_j(I_i)$ is said an immediate successor (predecessor) of $I_i(I_j)$.$P(I_i)$ is used to denote the set of predecessors of I_i and $S(I_i)$ the set of successors of I_i (not necessarily immediate in both sets).

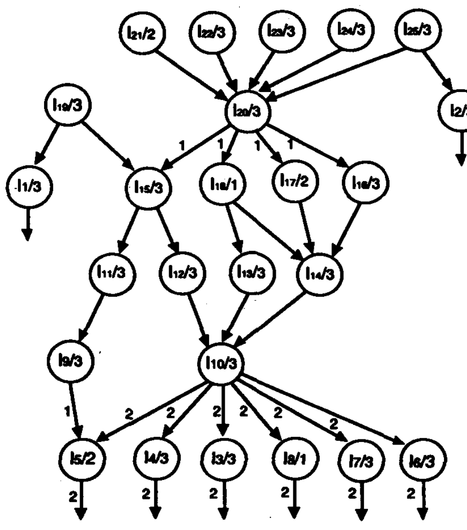


Fig. 1. A sample instruction DAG.

The function $\lambda: U \rightarrow \{1, 2, \cdots\}$ is said a *feasible schedule* of a given $(U, \prec, Pf, D)$ on a given $(k, M, \mathcal{P}, z)$ if $\lambda()$ satisfies the following conditions:

1) $\forall I_i, I_j \in U$, if $I_i \prec I_j$, then $\lambda(I_i) + D(I_i) < \lambda(I_j)$.
2) $\forall t \geq 1$ and $i \leq k$, $|\{I \in U | \lambda(I) = t \text{ and } Pf(I) = i\}| \leq m_i$.

$\lambda(I_i) = t_i$ means instruction I_i is scheduled in time slot t_i. Condition 1 stipulates that a schedule should obey the precedence relationships. Condition 2 stipulates that a schedule should not use more processors than are available. Fig. 2 shows a schedule for the DAG in Fig. 1. None of the successors of I_i can be issued until $\lambda(I_i) + D(I_i) + 1$. Thus we define $\lambda(I_i) + D(I_i)$ to be the *completion time* of I_i. The time slots $(\lambda(I_i), \lambda(I_i) + D(I_i))$ are said to be the *scope* of I_i. The length of a feasible schedule is defined to be $\max\{\lambda(I) + D(I)\}, \forall I \in U$. The optimization scheduling problem associated with the models given above is: given a $(k, M, \mathcal{P}, z)$ and a $(U, \prec, Pf, D)$, among all feasible schedules, find the schedule with a minimum length. This problem is NP-hard when $k > 1$ or $z > 1$ and $\prec$ defines an arbitrary acyclic graph, since it is known to be NP-complete when $(k = 2, M = (m_1 = 1, m_2 = 1), \mathcal{P}, z = 0)$ [14] or $(k = 1, M = (m_1 = 1), \mathcal{P}, z = 2)$ [10].

	1	2	3	4	5	6	7	8	9	10	11	12				
P_{11}					I_{18}					I_8						
P_{21}	I_{21}				I_{17}					I_5						
P_{31}	I_{25}	I_{22}	I_{20}		I_{16}	I_{14}	I_{10}			I_4						
P_{32}	I_{24}	I_2			I_{15}	I_{13}	I_9			I_3						
P_{33}	I_{23}	I_1				I_{12}				I_6						
P_{34}	I_{19}					I_{11}				I_7						

Fig. 2. A schedule for the sample DAG.

A schedule is said to be a greedy schedule if it satisfies the Principle of the Greedy Schedule. *If processor P_{ij} is idle at time t, then there is no instruction issued at a time later than t that could have been issued by P_{ij} at time t.* It can be easily verified that an optimal schedule can always be transformed into a greedy schedule. Thus our search for an optimal schedule is narrowed down to a search among greedy schedules.

Results relevant to our work are found in optimal UET task scheduling [15]. In [15], dominance and equivalence relations between tasks were first defined. These two relations were then used in a dynamic programming scheme. However, in the experiment in [15], the algorithm was effective only for small cases, partially owing to the limited memory capacity then available, but mainly because the algorithm could not reduce the solution space very much. Similar work was done by Yang *et al.* [16]. Yang *et al.* proposed a branch and bound scheme for solving UET schedules with time profiles. They provided five rules, two of which were termination rules and three of which were heuristic rules. The termination rules were used to check whether a partial schedule was an optimal schedule; if not, exploration of that schedule was stopped. The heuristic rules were used to select the next partial schedule for exploration.

The optimization scheme for UET task scheduling cannot be applied to superscalar instruction scheduling, because of the absence of the type and delay cycle features. Our optimization scheme is based on a solution tree used to keep track of all partial schedules. Suppose the DAG in Fig. 1 is to be executed on an $(m_1 = 1, m_2 = 1, m_3 = 4)$ architecture. A solution tree for finding the optimal schedule for the DAG in Fig. 1 is shown in Fig. 3. At the top of the solution tree is the root node. Let us call the nodes on the solution tree the *S-nodes*. An S-node, $Q_i = \{I_{i1}, I_{i2}, \cdots, I_{in}\}$, denotes the instructions executed in parallel in a time slot. The root is said to be at level 0, all of its children at level 1, and so on. Instructions $(I_{25}, I_{24}, I_{23}, I_{22}, I_{21}, I_{19})$ are the only ready instructions (RI) that can be executed at the very beginning. Since I_{21} is the only type 2 instruction, it can be executed immediately. The other five instructions compete for four type 3 processors. So level 1 of the solution tree consists of five S-nodes. By continuing in this manner, a complete solution tree can be built. Each path from the root to a leaf forms a feasible schedule. All the feasible schedules as well as an optimal schedule can be found if we traverse the entire solution tree.

The solution tree depicted in Fig. 3 is the most basic approach to obtaining an optimal schedule. This basic approach is however not efficient. Our effort in the optimal instruction scheduling will be directed toward reducing the number of S-nodes as much and as early as possible.

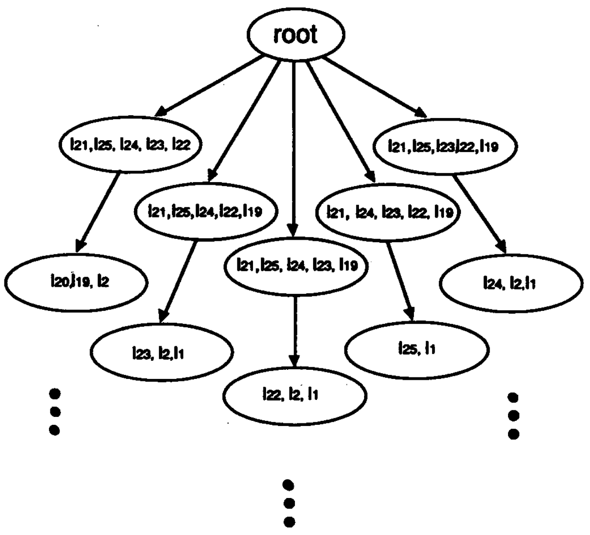


Fig. 3. Solution tree for the sample DAG.

III. *A Priori* Analysis

A. Dominance and Equivalence Relations Between Instructions

The basic solution tree scheme illustrated above enumerates all possible scheduling sequences of instructions. If, besides the precedence constraints that stipulate the scheduling sequences, more relations can be found to confine the scheduling, then the solution tree can be pruned. We hence define the natural dominance and equivalence relations as follows:

Definition 1) Natural Dominance Relation: If I_i can always be scheduled no later than I_j in an optimal schedule, then I_i is said to dominate I_j.

Definition 2) Natural Equivalence Relation: If I_i and I_j can always be interchanged in a schedule without affecting its length, then I_i is said to be equivalent to I_j.

In light of these definitions, the precedence relation is a kind of natural dominance relation.

However, to find all the natural dominance and equivalence relations among instructions is an intractable task, since it is necessary that the optimization problem be resolved beforehand. Nevertheless, some dominance and equivalence relations and, moreover, semi-dominance and semi-equivalence relations, can be found without carrying out any scheduling tasks.

Definition 3) Equivalence Relation on Instructions: If $I_i, I_j \in U$ and $Pf(I_i) = Pf(I_j)$, if $S(I_i) = S(I_j)$ and $P(I_i) = P(I_j)$ and $D(I_i) = D(I_j)$, then I_i is said to be equivalent to I_j, denoted by $I_i\,E\,I_j$ or $I_j\,E\,I_i$.

Definition 4) Semi-Equivalence Relation on Instructions: If $I_i, I_j \in U$ and $Pf(I_i) = Pf(I_j)$, if $S(I_i) = S(I_j)$ and $P(I_i) \neq P(I_j)$ and $D(I_i) = D(I_j)$, then I_i is said to be semi-equivalent to I_j, denoted by $I_i\,SE\,I_j$ or $I_j\,SE\,I_i$.

Definition 5) Dominance Relation on Instructions: Let $I_i, I_j \in U$ and $Pf(I_i) = Pf(I_j)$ and not $I_i\,E\,I_j$ nor $I_i\,SE\,I_j$. Then I_i is said to dominate I_j, denoted by $I_i\,D\,I_j$, if $S(I_i) \supseteq S(I_j)$ and $P(I_i) \subseteq P(I_j)$ and $D(I_i) \geq D(I_j)$.

Definition 6) Semi-Dominance Relation on Instructions: Let $I_i, I_j \in U$ and $Pf(I_i) = Pf(I_j)$, and not $I_i\, E\, I_j$ nor $I_i\, SE\, I_j$ nor $I_i\, D\, I_j$. Then I_i is said to semi-dominate I_j, denoted by $I_i\, SD\, I_j$, if I_i, I_j satisfy one of the following conditions:

1) $S(I_i) \supseteq S(I_j)$ *and* $D(I_i) \geq D(I_j)$.
2) $S(I_j) = \emptyset$ *and* $L(I_i) \geq D(I_j) + 1$.

All these relations are defined on instructions of the same type, since instructions of different types do not compete for the same processor. It is easy to see that the D and SD relations are transitive and nonsymmetric; the E relation is transitive, symmetric, and reflexive; and the SE relation is symmetric only. $I_i\, SE\, I_j$ and $I_j\, SE\, I_k$ do not imply that $I_i\, SE\, I_k$, since it is possible that $I_i\, E\, I_k$. Taking the DAG in Fig. 1 as an example, $I_{24}\, E\, I_{23}$, $I_{23}\, E\, I_{22}$, $I_{25}\, D\, \{I_{24}, I_{23}, I_{22}, I_{21}\}$, $I_{10}\, SD\, I_9$, $I_2\, SE\, I_1$, and $I_{12}\, SE\, I_{13}$.

It can easily be checked that when $I_i\, D\, I_j$, I_i and I_j comply with the natural dominance relation, that is, I_i is scheduled no later than I_j is in an optimal schedule. If otherwise, we can always obtain another schedule of no longer length by interchanging the positions of I_i and I_j. The semi-dominance relation does not possess this good property. The predecessors of the two intervolved instructions are different, and we cannot freely interchange I_i and I_j in a schedule only if $I_i\, SD\, I_j$. However, the semi-dominance relation still facilitates the reduction of the solution tree, though. The use of the semi-dominance relation is that if $I_i\, SD\, I_j$ and I_i and I_j are in RI simultaneously, then we should schedule I_i before I_j.

The meaning of the equivalence relation is intuitive. If $I_i\, E\, I_j$, then I_i and I_j will be in RI simultaneously. Whether I_i is scheduled before or after I_j makes no difference to the remainder of the schedule. The characteristics of the semi-equivalence relation are similar to those of the semi-dominance relation. If $I_i\, SE\, I_j$, we cannot interchange the positions of I_i and I_j in any arbitrary schedule. But if I_i and I_j are in RI simultaneously, then scheduling either I_i or I_j first will have the same effect on the remainder of the schedule.

B. Dominance and Equivalence Relations Between S-Nodes and Partial Schedules

We now derive the dominance and equivalence relations between S-nodes. Since instructions are typed, an S-node Q_i containing concurrently executed instructions is composed of *subS-nodes,* denoted by $Q_i = (q_i^1, q_i^2, \cdots, q_i^k)$, where q_i^j denotes the set of type j instructions in Q_i. Thus $q_i^j = (I_1^{q_i}, I_2^{q_i}, \cdots, I_{m_j}^{q_i})$, and $Pf(I_x^{q_i}) = j$, for $1 \leq x \leq m_j$. Dominance and equivalence relations can be defined on q_i^j and Q_i as follows:

Definition 7) Equivalence Relation on SubS-Nodes: Given two sub$S$-nodes q_1^n and q_2^n, q_1^n is equivalent to q_2^n, denoted by $q_1^n\, E\, q_2^n$, if $I_i^{q_1} = I_i^{q_2}$ or $I_i^{q_1}\, E\, I_i^{q_2}$ or $I_i^{q_1}\, SE\, I_i^{q_2}$, for all i.

Definition 8) Dominance Relation on SubS-Nodes: Given two sub$S$-nodes q_1^n and q_2^n, q_1^n dominates q_2^n, denoted by $q_1^n\, D\, q_2^n$, if not $q_1^n\, E\, q_2^n$ and $(I_i^{q_1} = I_i^{q_2}$ or $I_i^{q_1}\, D\, I_i^{q_2}$ or $I_i^{q_1}\, SD\, I_i^{q_2})$, for all i.

Definition 9) Equivalence Relation on S-Nodes: Let Q_1 and Q_2 be two S-nodes under the same parent in a solution tree. Q_1 is equivalent to Q_2, denoted by $Q_1\, \underline{E}\, Q_2$, if $q_1^i\, E\, q_2^i$ for $1 \leq i \leq k$.

Definition 10) Dominance Relation on S-Nodes: Let Q_1 and Q_2 be two S-nodes under the same parent in a solution tree. Q_1 dominates Q_2, denoted by $Q_1\, \underline{D}\, Q_2$, if not $Q_1\, E\, Q_2$ and $(q_1^i\, E\, q_2^i$ or $q_1^i\, D\, q_2^i)$, for $1 \leq i \leq k$.

For example, for the S-nodes in Fig. 3, we have

$$(I_{21}, I_{25}, I_{24}, I_{23}, I_{19})\, \underline{E}\, (I_{21}, I_{25}, I_{24}, I_{22}, I_{19}),$$

since $I_{23}\, E\, I_{22}$; and

$$(I_{21}, I_{25}, I_{24}, I_{23}, I_{19})\, \underline{D}\, (I_{21}, I_{24}, I_{23}, I_{22}, I_{19})$$

since $I_{25}\, D\, I_{22}$.

Furthermore, the dominance and equivalence relations between partial schedules can be also defined. Before doing so, however, the concept of *live instructions* must be introduced. Taking Fig. 2 as an example, instruction I_{10} has a delay of two cycles and is scheduled in time slot 7. We say I_{10} is alive in time slots (7, 8, 9), since its scope covers these slots and any successors of I_{10} cannot be scheduled in these slots. Let $S1$ be a schedule or a partial schedule. Instructions that are alive in time slot i of $S1$ are the instructions whose scopes cover time slot i, denoted by $Live(S1, i)$. Let $S1$ denote the aggregation of the instructions in $S1$. The dominance and equivalence relations between partial schedules are as following:

Definition 11) Equivalence Relation on Partial Schedules: Let $S1$ and $S2$ be two partial schedules with the same length j. $S1$ is said to be equivalent to $S2$, denoted by $S1\underline{E}\, S2$ or $S2\underline{E}S1$, if $S1$ and $S2$ satisfy the following conditions:

1) $|Live(S1, j)| = |Live(S2, j)|$ *and there is a complete matching from* $Live(S1, j)$ *to* $Live(S2, j)$, *such that* $\lambda(I_{s1}) = \lambda(I_{s2})$ *and* $(I_{s1} = I_{s2}$ *or* $I_{s1}\, E\, I_{s2}$ *or* $I_{s1}\, SE\, I_{s2})$, $I_{s1} \in Live(S1, j)$ *and* $I_{s2} \in Live(S2, j)$.
2) $|U - \{S1\}| = |U - \{S2\}|$ *and there is a complete matching from* $U - \{S1\}$ *to* $U - \{S2\}$, *such that* $I_{s2} = I_{s1}$ *or* $I_{s1}\, E\, I_{s2}$, *where* $I_{s1} \in U - \{S1\}$ *and* $I_{s2} \in U - \{S2\}$.

Definition 12) Dominance Relation on Partial Schedules: Let $S1$ and $S2$ be two partial schedules with lengths i and j, respectively, where $i \leq j$ and not $S1\, \underline{E}\, S2$. $S1$ is said to dominate $S2$, denoted by $S1\, \underline{D}\, S2$, if $S1$ and $S2$ satisfy the following conditions:

1) *There is a complete matching from* $Live(S1, i)$ *to* $Live(S2, j)$, *such that* $\lambda(I_{s1}) = \lambda(I_{s2})$ *and* $(I_{s1} = I_{s2}$ *or* $I_{s1}\, E\, I_{s2}$ *or* $I_{s1}\, SE\, I_{s2})$, $I_{s1} \in Live(S1, i)$ *and* $I_{s2} \in Live(S2, j)$.
2) $|U - \{S1\}| \leq |U - \{S2\}|$, *and there is a complete matching from* $U - \{S1\}$ *to* $U - \{S2\}$, *such that* $I_{s1} = I_{s2}$ *or* $I_{s1}\, E\, I_{s2}$ *or* $I_{s1}\, D\, I_{s2}$, *where* $I_{s1} \in U - \{S1\}$ *and* $I_{s2} \in U - \{S2\}$.

Definitions 9) and 10) may be regarded as special cases of Definitions 11) and 12), respectively. However, Definitions 9) and 10) are concerned with S-nodes under the same parent, and the conditions in these definitions are looser than those in Definitions 11) and 12). Another reason to separate Definitions 9) and 10) from Definitions 11) and 12) is that the S-nodes under the same parent are generated simultaneously, so we can avoid generating unnecessary S-nodes quicker. Taking again Fig. 3 as an example, the partial schedule ending with S-node (I_{22}, I_2, I_1) is equivalent to the partial schedule ending with (I_{23}, I_2, I_1) but dominates the partial schedule ending with (I_{25}, I_1).

These definitions are defined hierarchically and are based only on relations among instructions. Our objective in defining these relations is to find those S-nodes and partial schedules that cannot lead to an optimal schedule during construction of the solution tree. It is proved below that dominated S-nodes need not be generated in the solution tree, since such S-nodes cannot lead to a solution better than the solutions emanating from their dominating S-nodes. And only one of a pair of equivalent S-nodes needs to be generated, since both will lead to schedules of the same length. Similarly, dominated partial schedules need not be continued, and it is sufficient to continue only one of a pair of equivalent partial schedules.

The following four lemmas define the unnecessary S-nodes and partial schedules. The proofs are not very difficult and are omitted here. Interested readers may refer to [17].

Lemma 1: The dominated S-nodes in a solution tree need not be generated.

Lemma 2: Let Q_1 and Q_2 be two S-nodes such that $Q_1 \underline{E} Q_2$. Then either Q_1 or Q_2 need not be generated.

Lemma 3: Let $S1$ and $S2$ be two partial schedules such that $S1 \underline{E} S2$. Then either $S1$ or $S2$ need not be continued.

Lemma 4: Let $S1$ and $S2$ be two partial schedules such that $S1 \underline{D} S2$. Then $S2$ need not be continued.

IV. Implementation

In this section, we first present a basic generation algorithm which generates all legal S-nodes from RI. Then we modify the algorithm so that it generates only nondominated S-nodes, and further modify it to eliminate equivalent S-nodes. For partial schedule termination, each time an S-node is generated, we check it against the S-nodes previously generated on the same level. From the definitions, the checking involves two matching procedures on live instructions and unscheduled instructions, respectively. The complexity of the matching procedure is normally the cube of the input size [18]. But we will show that when S-nodes are generated by our algorithm, the matching procedure becomes quite straightforward.

A. Basic S-Node Generation Algorithm

Let RI_i denote the type-i instruction subset of RI and $\{q^i\}$ denote the m_i-element subset of RI_i. All the legal children S-nodes for the next time slot can be obtained by taking the Cartesian product of the $\{q^i\}$s, $RI = \{q^1\} \times \{q^2\} \times \cdots \{q^k\}$.

The basic sub$S$-node generation algorithm is to find the m_i-element subsets from RI_i, and generate S-nodes by carrying out a Cartesian product of the sub$S$-nodes. Although there exists an algorithm for finding such subsets [19], we use a new approach, which can then be refined to serve our final goal. The basic algorithm, called Algorithm A, generates all the m_i-element subsets from RI_i by constructing a *generation tree.*

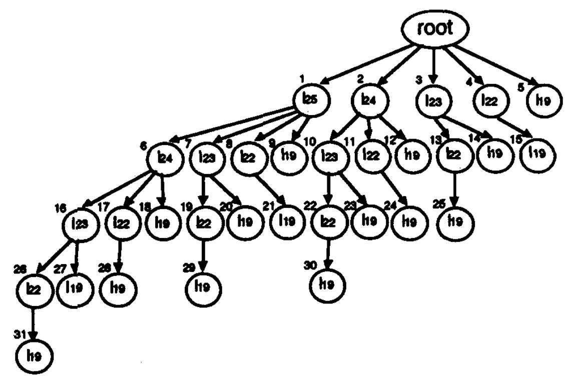


Fig. 4. Generation tree of $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$ by Algorithm A.

Consider instructions $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$ in Fig. 1; these are type 3 instructions that are ready at the very beginning. Taking these instructions as its input, Algorithm A constructs the generation tree shown in Fig. 4. Since an instruction may appear several times in the tree, we place an index in the upper left corner of each node to distinguish the instructions. Each path from the root consisting of i nodes forms a sub$S$-node of i instructions. All the paths originating from the root and consisting of 4 ($m_3 = 4$ in this case) nodes form the $\{q^3\}$ set for the next time slot.

Let Y denote a set of nodes in the generation tree, $LS(Y)$ denote the set of left siblings of the elements in Y in the generation tree, and $R(I^*)$ denote the ancestor of I^* in the generation tree. For example, in Fig. 4, $LS(11) = \{10\}$ and$R(11) = \{2\}$, where the numbers represent the indexes of the generation tree nodes. Algorithm A constructs the tree by following a rule of thumb: each node I^* in the tree will have other instructions as its children except for the nodes in the set of $R(I^*)$ and $LS(R(I^*) + I^*)$.

Algorithm A

```
/* input: instructions in $RI_i$ */
/* output: a generation tree of type $i$ */
1   push-stack(root);
2   **while** (stack not empty) **do**
3       $x$ = pop-stack;
4       $W = LS(R(x) + x)$;
5       $E = RI_i - R(x) - W - x$;
6       **for** each $e$ in $E$
7           link $e$ to $x$ as a child;
8           push-stack($e$);
9       **endfor**
10  **endwhile**
```

There are a total of five sub$S$-nodes, $(I_{25}, I_{24}, I_{23}, I_{22})$, $(I_{25}, I_{24}, I_{23}, I_{19})$, $(I_{25}, I_{23}, I_{22}, I_{19})$, $(I_{25}, I_{24}, I_{22}, I_{19})$, and $(I_{24}, I_{23}, I_{22}, I_{19})$, obtained from Fig. 4. These five sub$S$-nodes combined with I_{21} (the only type 2 instruction ready for execution at the very beginning) form the five legal S-nodes at the first level in Fig. 3.

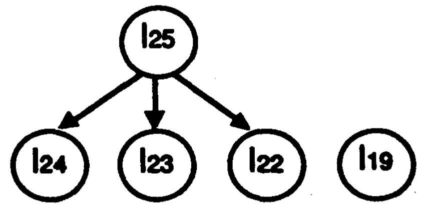


Fig. 5. Dominance graph of instructions $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$.

The generation tree can be constructed in a depth-first or breadth-first (by replacing the stack in the algorithm with a queue) manner. The computations of $LS(R(x))$ and $R(x)$ on lines 4 and 5 are not difficult, if some links between nodes are maintained. It will be seen in Lemma 5 that each node in the generation tree represents a subset of RI_i, thus the complexity of Algorithm A is $O(2^{|RI_i|})$. This is not surprising, since the instruction scheduling problem is NP-hard.

The generation tree in Fig. 4 need not be fully constructed. If the number of instructions in RI_i is larger than m_i, each path is terminated as soon as its length is equal to $m_i (m_3 = 4$ in this case). Those paths whose lengths equal to m_i form the required sub$S$-nodes. The shorter paths are discarded, since sub$S$-nodes obtained from these paths violate the Principle of the Greedy Schedule and may not lead to an optimal schedule. The tree constructed by the algorithm is not unique, since we did not impose any order on the instructions in the set E. A number of characteristics of the generation tree can be identified:

1) All the right siblings are also children;
2) None of the left siblings are children;
3) No two paths or sub-paths from the root are identical;
4) All legal sub$S$-nodes can be derived from the generation tree.

Characteristics 1 and 2 follow directly from Algorithm A. Characteristic 3 is a consequence of 1 and 2. Characteristic 4 can be proved by showing that all the subsets of RI_i can be derived from a fully constructed generation tree. That is, all the paths from the root with length j form the j-element subsets of RI_i.

Lemma 5: All subsets of RI_i containing j elements can be derived from the generation tree by traversing each path j steps.

Proof: Refer to [17].

Corollary 1: Let instructions $\{I_1, I_2, \cdots, I_k\}$ be siblings in a generation tree and let I_1 be the leftmost node among them. Then there exists a path in the tree starting with I_1 that is labeled with $\{I_1, I_2, \cdots, I_k\}$.

This corollary does not require that $\{I_1, I_2, \cdots, I_k\}$ be all of the children of a parent node; The corollary still holds when $\{I_1, I_2, \cdots, I_k\}$ is a subset of the children.

B. Nondominated S-Node Generation Algorithm

Since Algorithm A generates all the m_i-element subsets, it is feasible only when there are no useful relations among the instructions. We will now modify Algorithm A taking the dominance relations into consideration. The revised algorithm,

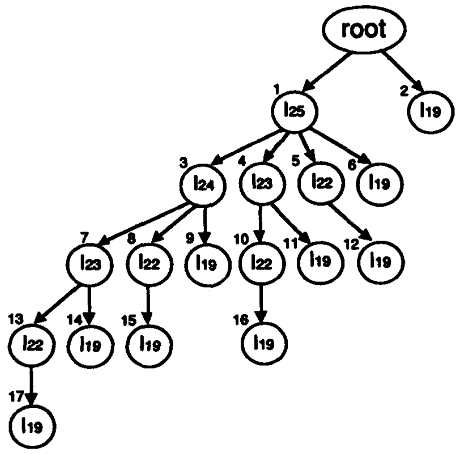


Fig. 6. Generation tree of $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$ by Algorithm B.

which we call Algorithm B, does not generate dominated sub$S$-nodes.

The D and SD relations between the instructions in RI_i can be represented by a DAG called a *dominance graph.* In this graph, a directed edge exists between a dominant instruction and its dominated or semi-dominated instruction. Fig. 5 shows the dominance graph of $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$.

The dominance graph facilitates the generation of nondominated sub$S$-nodes. In Algorithm B, a dominated instruction cannot be linked to the generation tree except to a path on which all its dominant instructions are present. Let $DM(X)$ denote the set of instructions in RI_i such that all of their dominant instructions in RI_i are in X. For example $DM(I_{25}) = (I_{24}, I_{23}, I_{22})$.

Algorithm B

```
/* input: instructions in $RI_i$ */
/* output: a generation tree of type $i$ */
1   push-stack(root);
2   Nd = nondominated instructions in $RI_i$
3   **while** (stack not empty) **do**
4       $x$ = pop-stack;
5       $W = LS(R(x) + x)$;
6       $E = Nd + DM(R(x) + x) - W - R(x) - x$;
7       **for** each $e$ in $E$
8           link $e$ to $x$ as a child;
9           push-stack($e$);
10      **endfor**
11  **endwhile**
```

Fig. 6 shows the generation tree constructed by Algorithm B for $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$. There are only four sub$S$-nodes, and none of them dominats the others. However, the complexity of Algorithm B is also $O(2^{|RI_i|})$.

Characteristics 1, 2, and 3 of Algorithm A are inherited by Algorithm B. Characteristic 4 is true only when there is no dominance relation between instructions in RI_i. Corollary 1 is also true for Algorithm B, since it is a result of characteristics

1 and 2. The generation tree constructed by Algorithm B has two additional characteristics:

5) Not all instructions in RI_i are linked directly to the root node. Only instructions which have no dominator in RI_i are.
6) Dominated instructions appear immediately after all of their dominators have appeared on the same path, such as nodes $\{3, 4, 5\}$ in Fig. 6.

Also, like the tree constructed by Algorithm A, the tree constructed by Algorithm B is not unique.

Lemma 6: All but dominated sub$S$-nodes can be derived from the tree constructed by Algorithm B.

Proof: We first prove that dominated sub$S$-nodes will not be derived from the tree. From the definition of the dominance relation on sub$S$-nodes, if q_j^i is a dominated sub$S$-node, then there must be an instruction, say $I_y^{q_j}$, in q_j^i, for which the dominant instruction, say $I_x^{q_j}$, is not in q_j^i. But Algorithm B never produces this kind of sub$S$-node. Since we treat the dominance relation as similar to the precedence relation, an instruction cannot be linked to the tree if not all of its dominant instructions are on the same path.

We also prove that the algorithm is complete, i.e., that all the nondominated sub$S$-nodes will be derived. Suppose there is a nondominated sub$S$-node $q_j^i = \{I_1^{q_j}, I_2^{q_j}, \cdots, I_n^{q_j}\}$ that cannot be derived from the tree. First we partition the instructions in q_j^i into two disjoint subsets, q_A^i and q_B^i. q_A^i contains all the nondominated instructions in q_j^i, and q_B^i contains the remaining dominated instructions in q_j^i. Since the instructions in q_A^i are not dominated, all of them are linked to the root node. Suppose that $I_1^{q_j}$ is the leftmost sibling among them. Consider the children of $I_1^{q_j}$. Since all right siblings of a node will be also its children, the children of $I_1^{q_j}$ are $q_A^i - I_1^{q_j} + DM(I_1^{q_j})$ $(DM(I_1^{q_j}) \subseteq q_B^i)$. Suppose again without loss of generality that $I_2^{q_j}$ is the leftmost sibling among $q_A^i - I_1^{q_j} + DM(I_1^{q_j})$. Then the children of $I_2^{q_j}$ are the set $q_A^i - \{I_1^{q_j}, I_2^{q_j}\} + DM(\{I_1^{q_j}, I_2^{q_j}\})$. If we continue in this manner, since all the dominant instructions of q_B^i are in q_A^i, all of the instructions in q_B^i will finally be activated by the function $DM()$ and thus will be linked. So we can find a path starting from the root labeled with only the instructions in q_j^i. □

In the above proof, we do not specify any particular order in which the instructions are linked to their parent, so Lemma 6 also proves that all the trees constructed by Algorithm B are equivalent in generating sub$S$-nodes. However, within the trees, equivalent sub$S$-nodes still exist. The three sub$S$-nodes $(I_{25}, I_{24}, I_{23}, I_{19}), (I_{25}, I_{24}, I_{22}, I_{19})$; and $(I_{25}, I_{23}, I_{22}, I_{19})$ derived from the tree in Fig. 6 are actually equivalent, because I_{24}, I_{23} and I_{22} are equivalent instructions. So Algorithm B needs further improvement.

C. Eliminating Equivalent Sub$S$-Nodes

We start from the generation tree in Fig. 6. Consider nodes 3, 4, and 5, which represent instructions I_{24}, I_{23} and I_{22}, respectively. Since $I_{24}\, E\, I_{23}$, and I_{24} is the left sibling of I_{23}, all the right siblings of I_{23} are also right siblings of I_{24}. So the children of node 3 encompasses the children of node 4.

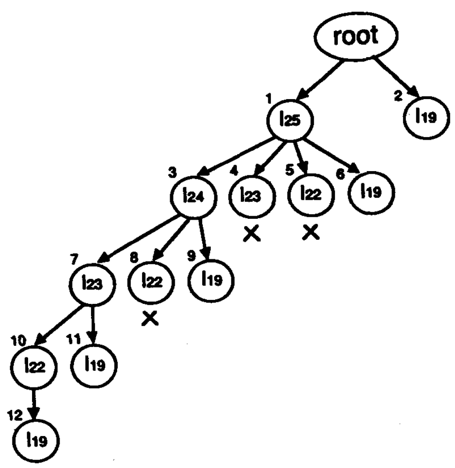


Fig. 7. Generation tree of $(I_{25}, I_{24}, I_{23}, I_{22}, I_{19})$ by Algorithm C.

From Corollary 1, for any path starting from node 4, there is a corresponding path starting from node 3 containing the same set of instructions, except that the first instruction is replaced by I_{24} instead of I_{23}. Thus, the exploration of node 4, and also node 5, should be terminated to eliminate equivalent sub$S$-nodes. We modify Algorithm B by adding one more constraint: the exploration of equivalent instructions in the set E on line 7 of Algorithm B should be stopped. The resulted algorithm, named Algorithm C, is as follows:

Algorithm C

```
/* input: $RI_i$, dominance graph and equivalence
        relations */
/* output: a reduced generation tree of type $i$ */
1 push-stack(root);
2 Nd = nondominated instructions in $RI_i$;
3 while (stack not empty) do
4     $x$ = pop-stack;
5     $W = LS(R(x) + x)$;
6     $E = Nd + DM(R(x) + x) - W - R(x) - x$;
7     terminate redundant equivalent or semi-equivalent
      instructions in E;
8     for each instruction e in E
9         link e to x as a child;
10        push-stack(e);
11    endfor
12 endwhile
```

Lemma 7: All but nondominated and nonequivalent sub$S$-nodes can be derived from the tree generated by Algorithm C.

Proof: The proof follows directly from the argument above. □

The final generation tree is given in Fig. 7. Only two sub$S$-nodes $(I_{25}, I_{24}, I_{23}, I_{22})$ and $(I_{25}, I_{24}, I_{23}, I_{19})$ are obtained from the tree. The nodes marked with '×' are terminated owing to equivalence (or semi-equivalence) relations.

D. Eliminating Dominated and Equivalent Partial Schedules

It is not likely that we can prevent the generation of all dominated and equivalent partial schedules, since these schedules may emanate from different parent S-nodes. From Definitions 11) and 12), we know that checking the dominance and equivalence conditions will involve two matching procedures on live instructions and unscheduled instructions, respectively. To formalize the matching problem, let us translate it into a bipartite graph matching problem. Let $(V1, V2, E)$ denote a bipartite graph. The matching problem in condition 2 of Definition 12) is translated to:

- $V1$ denotes the set of instructions in $U - \{S1\}$.
- $V2$ denotes the set of instructions in $U - \{S2\}$.
- E is the set of edges. Let $I_{v1} \in U - \{S1\}$ and $I_{v2} \in U - \{S2\}$. $(I_{v1}, I_{v2}) \in E$ if $I_{v1} = I_{v2}$ or $I_{v1}\, E\, I_{v2}$ or $I_{v1}\, D\, I_{v2}$.

Condition 2 of Definition 12) is satisfied if and only if there is a complete matching on $V1$ and $V2$. Lawler [18] presented an $O(|V1|^2|V2|)$ algorithm to find such a matching if it exists. However, the matching task can be reduced to $O(|V1|^2)$ if the S-nodes are generated by Algorithm C. The equivalence relation between instructions is reflexive, symmetric and transitive, so equivalent instructions will form an equivalence class. The instructions in $V1$ and $V2$ can thus be divided into several equivalence classes. If $I_{v1} \in V1, I_{v2} \in V2$ and $(I_{v1}, I_{v2}) \in E$, then I_{v1} also has edges to all the other instructions which belong to the same equivalence class of I_{v2}. Thus if we neglect the edges originates from dominance relations (though they are allowed), the instruction matching task is reduced to an equivalence class matching task. To check whether a complete matching exists, we need only check that the number of instructions in each matched equivalence class are equal.

We now prove that when S-nodes are generated by Algorithm C, if there is a complete matching on $V1$ and $V2$, then this matching contains only equivalence relations.

Lemma 8: When S-nodes are generated by Algorithm C, if condition 2 of Definition 12) is satisfied, then the matching contains only equivalence relations.

Proof: We prove this lemma by contradiction. Suppose there is a matching containing dominance relation. Assume the dominance relation is: I_a matches $I_b, I_a \in V1, I_b \in V2$, and $I_a\, D\, I_b$. Since $I_a\, D\, I_b$, from Algorithm C, I_b must also be in $V1$. Consider the matched instruction of I_b in $V2$, and assume it to be I_c. Then $I_b\, E\, I_c$ or $I_b\, D\, I_c$. In either case, we have $I_a\, D\, I_c$, thus I_c is also in $V1$, from Algorithm C. If we go on in this manner, an instruction in $V1$ which has no matched instruction in $V2$ can always be found. So the matching does not exist. We conclude that if there is a matching to satisfy condition 2 of Definition 12), this matching contains no dominance relation.

Next, consider the other conditions in Definition 11) and Definition 12). For the first two conditions in Definition 11) and Definition 12), the two sets can also be divided into equivalence classes, with each class formed by instructions that are scheduled in the same time slot and have equivalence or semi-equivalence relations with each other. Condition 2 of Definition 11) can also be treated in a similar way. Thus the matching task again becomes an equivalence class matching problem. So we conclude that to check the conditions in Definitions 11) and 12) it suffices to check whether the number of instructions in the matched equivalence classes is equal or not, and the complexity of this task is at most a square of the input size.

E. Optimization Scheduling Algorithm

The optimization scheme basically involves constructing the solution tree using Algorithm C.

Optimization Scheduling Algorithm:

```
/* input: instructions to be scheduled with precedence
      relation */
/* output: solution tree */
1 push-stack(root)
2 while (stack not empty) do
3     $x$ = pop-stack;
4     compute RI;
5     for $i = 1$ to $k$
6         $S_i$ =sub$S$-nodes generated from $RI_i$ by
          Algorithm C;
7     endfor
8     S-set = $S_1 \times S_2 \times \cdots S_k$;
9     for each S-node s in S-set
10        link $s$ to $x$ as a child;
11        check if the partial schedule emanating
          from $s$ is dominated or equivalent to
          other schedules
12    endfor
13 endwhile
```

Owing to the NP-complete nature of the problem, the complexity of the scheme is basically exponential. The actual complexity will depend on the dependencies between instructions, and it is impossible to give a detailed complexity analysis here. The final solution tree for the sample DAG is illustrated in Fig. 8. The partial schedule on the right is terminated, because it is dominated by the schedule on the left. The S-nodes which are trimmed by Algorithm C are not shown in the figure.

V. Experiment and Results

The proposed algorithm was run on an HP 9000/425 workstation. We used a random procedure to generate test data, instead of using real programs. This decision is based on the following reasons. First, the scope of this study is on a basic block, such as in local scheduling, or a program trace, such as in global scheduling, but not on the entire program. Secondly, there is no common agreement as to what a typical program trace should be, and a good instruction scheduler should react to different input code properly. And thirdly, it is important that we capture the knee points of the various performance curves in the experiment, so that the behavior and performance of the different schedulers can be thoroughly

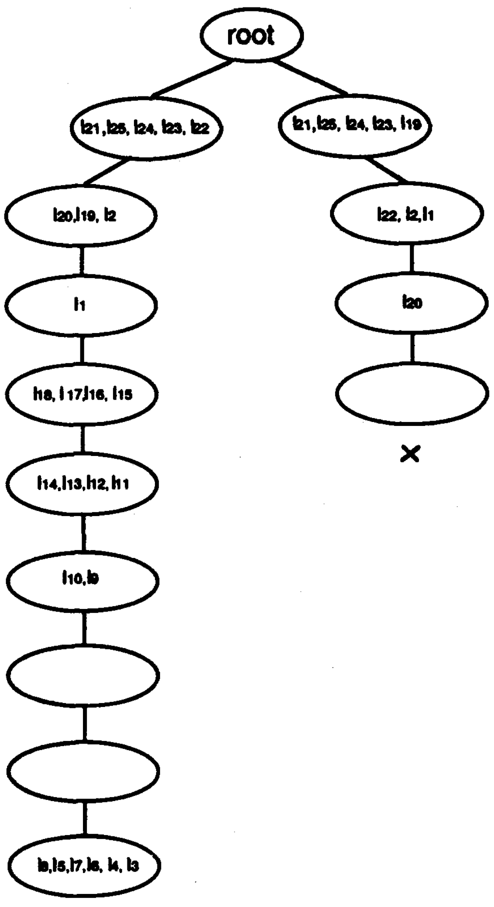


Fig. 8. Final solution tree constructed by the proposed scheme.

tested and analyzed The parameters involved in the random procedure include:

- k: the number of processor types;
- m_i: the number of processors of each type i;
- n: the number of instructions to be scheduled;
- the distribution of instructions of all types;
- the dependencies between instructions.

In our experiments, we selected $k = 2$ and $k = 4$, because no contemporary superscalar architecture employs more than four different types of processors. Moreover, it is shown in [17] that the more types of processor a superscalar architecture employs, the worse the worst-case scheduling behavior becomes. When $k = 2$, there is one fixed and one floating-point processor; when $k = 4$, two additional types are branch processor and conditional-register processor. This configuration is the same as that of the IBM RS/6000.

When $k = 2$, we assume $m_1 = m_2 = 1$ and $m_1 = m_2 = 2$. For $k = 4$, we assume $m_1 = m_2 = m_3 = m_4 = 1$, or $m_1 = 4, m_2 = 2, m_3 = 1$ and $m_4 = 1$. We select these m_i values so that the summation of m_i in each case is a power of two.

We select $n = 20$ (number of instructions scheduled in a batch). Since this number is large enough to approximates that of a large program trace in general-purpose applications, and to reveal the advantages of our scheme. A random procedure is uesd to determine the type and delay of each instruction and the dependency relationships among instructions. An n by n adjacency matrix M is used to represent the dependency relationships among instructions. For each $M(i,j)$ entry, for $i > j$, a random number was generated. This random number was compared against a preset density value S, and $M(i,j) = 1$ if the random number was less than S, and $M(i,j) = 0$ otherwise. $M(i,j) = 1$ means that $I_i \prec I_j$. In our experiments, the variable S which controls the sparseness of the matrix was increased from 0.00 to 0.30 with a stride of 0.02. For the distribution of types and delays of instructions, we adopted data from Shiau and Chung [20]. They showed that fixed-point instructions account for about 83% of all instructions, floating-point 12%, branch 4%, and conditional-register 1%. A random number was used to assign the types of instructions.

It is known that the numbers generated by a random number generator reach a stable distribution after the generator is called a certain number of times. Thus, for each density value, five instances of adjacent matrices (DAGs) were created and run. The experimental results were taken from the average of the results of the five runs. Three programs were coded to run on the generated DAGs. The first program, Alg-C&PS, implements the proposed scheme. The second program, Alg-C, implements only Algorithm C. The third program, BF, uses an exhaustive, brute force manner to construct the solution tree.

The number of S-nodes generated and the time consumed by each program for each run are collected and processed. The number of S-nodes for that program in each case is calculated from the arithmetic mean of the five runs. For computation time, instead of comparing the absolute time consumed, it is more meaningful to show the speedups resulted from Alg-C&PS to Alg-C and BF. Let T_{1i}, T_{2i}, and T_{3i} denote the computation time of Alg-C&PS, Alg-C, and BF in each run, respectively. So the ratios T_{2i}/T_{1i} and T_{3i}/T_{1i} are the speedups of Alg-C&PS to Alg-C and BF, respectively. The harmonic means, instead of arithmetic means, of these speedups are calculated.

The programs are allowed to be aborted due to a shortage in memory (if the solution tree grows too large). In the experiment, Alg-C&PS was never aborted, and Alg-C was aborted only in a few cases. But BF was rarely completed. If a program was terminated, we assign default values for that run. The default value for the number of S-nodes is 10000, and the default computation time is 60 seconds, which is a little longer than the maximum computation time of any successful run. Although this assumption is very conservative, the efficiency of Alg-C&PS is still evident.

Tables I and II list the experimental results for $(m_1 = 1, m_2 = 1)$ and $(m_1 = 1, m_2 = 1, m_3 = 1, m_4 = 1)$ processor organizations. In these tables, a '-' entered for the number of S-nodes indicates that the program failed to complete execution in any of the five runs. In these cases, the relative speed was more than 300—a conservative estimate. One can see that Alg-C&PS performs far better than the other two programs. Some other interesting phenomena can

TABLE I
NUMBER OF S-NODES AND RELATIVE SPEED OF $(m_1 = 1, m_2 = 1)$.

	No. S-nodes			Relative speed		
Density	Alg-C&PS	Alg-C	BF	Alg-C&PS	Alg-C	BF
0.30	82	4472	-	1.0	29.1	-
0.28	63	6046	-	1.0	34.2	-
0.26	141	6689	-	1.0	61.8	-
0.24	137	8453	-	1.0	192.1	-
0.22	178	8246	-	1.0	138.8	-
0.20	203	8184	-	1.0	115.7	-
0.18	431	8394	-	1.0	135 .5	-
0.16	358	9757	-	1.0	153.0	-
0.14	238	-	-	1.0	-	-
0.12	499	-	-	1.0	-	-
0.10	472	-	-	1.0	-	-
0.08	155	-	-	1.0	-	-
0.06	161	9030	-	1.0	166.7	-
0.04	84	7161	-	1.0	103.2	-
0.02	57	3080	-	1.0	5.6	-
0.00	18	18	-	1.0	1.0	-

TABLE II
NUMBER OF S-NODES AND RELATIVE SPEED OF $(m_1 = 1, m_2 = 1, m_3 = 1, m_4 = 1)$

	No. S-nodes			Relative speed		
Density	Alg-C&PS	Alg-C	BF	Alg-C&PS	Alg-C	BF
0.30	44	1362	-	1.0	26.3	-
0.28	97	4166	-	1.0	6.1	-
0.26	145	4068	-	1.0	5.5	-
0.24	79	6697	-	1.0	11.6	-
0.22	276	7333	-	1.0	17.9	-
0.20	192	8718	-	1.0	51.3	-
0.18	119	9210	-	1.0	23.3	-
0.16	150	8153	-	1.0	141.0	-
0.14	166	7585	-	1.0	106.6	-
0.12	349	-	-	1.0	-	-
0.10	150	8465	-	1.0	128.8	-
0.08	273	9117	-	1.0	249.5	-
0.06	113	5145	-	1.0	44.9	-
0.04	124	5310	-	1.0	37.4	-
0.02	70	5412	-	1.0	34.8	-
0.00	23	23	-	1.0	1.0	-

also be observed in these tables. Taking Table I for example, we observed that the number of S-nodes in the columns Alg-C&PS and Alg-C increases as the density decreases but decreases when the density is very small. The reason is that when the density is high, there are many dependencies or dominance relations among instructions. So the number of ready instructions in each time slot is small. The solution tree tends to be narrow, and the number of S-nodes is small. As the density decreases, the number of ready instructions in each time slot increases. The solution tree becomes wide, and the number of S-nodes grows large. However, as the density becomes very small, the dominance relations diminish yet the equivalence relations increase. Alg-C&PS and Alg-C both can take advantage of equivalence relations, so the solution tree becomes narrow again, and the number of S-nodes becomes small. In the extreme case when $S = 0.00$, all the instructions with the same type and delay are equivalent instructions. The solution trees constructed by Alg-C&PS and Alg-C are almost a single chain. In contrast, BF fails to take this advantage, so it runs out of memory, even though the DAGs in these zero-density cases are very easy to schedule. More experimental results can be found in [17].

The second phenomenon we noticed in these tables is that the change in the values in each column is not strictly monotonic. This is because the DAG's are randomly generated. The density value can control only the number of edges in these DAG's, but not the way the dependencies are allocated, or the "shape" of the DAG's. However, it is known that not only the number of edges, but also the "shape" of the DAG's affect scheduling. This phenomenon will become less apparent or disappear if we were to generate thousands of DAG's in each case instead of only five. Here we only point out this phenomenon instead of concealing it by running more DAG's.

Alg-C&PS, as expected, produces far fewer S-nodes than the other two methods in most cases. We conclude that Alg-C&PS is efficient in both time and space required. Owing to the ability to generate only those S-nodes and partial schedules nearly absolutely needed, we believe that Alg-C&PS is practical to schedule real code, regardless of the number of instructions involved, and remains superior to the other methods.

VI. DISCUSSION

From Lemma 1, we learned that a scheduling algorithm should give scheduling priority to dominant instructions over their corresponding dominated instructions. If not, a less optimal schedule may result. Also we suggest that a scheduling algorithm give scheduling priority to semi-dominant instructions, since it is likely that semi-dominant instructions will be scheduled no later than their corresponding semi-dominated instructions in an optimal schedule.

The methodology described in the above sections can be easily extended to other scheduling problems, especially those where the computation times of the scheduled objects are identical. The first natural extension is to the UET task scheduling problem. Since UET tasks involve no type or delay features, the dominance and equivalence relations for UET tasks are simpler than those defined in Section III. The constraints on type and delay in Definition 3) through Definition 6) are thus unnecessary for UET task scheduling. Another application of our scheme is in UET task scheduling with a deadline [14]. In this problem, each task T_i is associated with a deadline $d(T_i)$. The scheduling criterion requires that each task be completed before its deadline. The dominance and equivalence relations on such tasks require one more constraint than pure UET tasks. The additional constraint is $d(T_i) = d(T_j)$, if T_i is equivalent or semi-equivalent to T_j; and $d(T_i) \leq d(T_j)$, if T_i dominates or semi-dominates T_j.

VII. CONCLUDING REMARKS

Dominance characteristics have been exploited to solve many scheduling problems. We believe that our approach is different from previous approaches in one very important respect. Previous approaches emphasized dominance relations only. Our scheme takes advantage of not only dominance but also semi-dominance, equivalence, and semi-equivalence relations. Moreover, we also define hierarchical dominance and equivalence relations on S-nodes and partial schedules. We propose an algorithm, named Algorithm C, that avoids the generation of dominated and redundant equivalent S-nodes. The optimization scheduling algorithm presented in Section IV–E also facilitates the checking of dominated partial schedules. Finally, our experiments show the efficiency of the proposed scheme.

References

[1] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 processor architecture," *IBM J. Res. Develop.*, vol. 34, pp. 23–36, 1990.

[2] N. Margulis, *i860 Microprocessor Architecture*. New York: McGraw-Hill, 1990.

[3] V. J. Rayward-Smith, "UET scheduling with unit interprocessor communication delays," *Discrete Appl. Math.*, vol. 18, pp. 55–71, 1987.

[4] M. Johnson, *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[5] H. S. Warren, Jr., "Instruction scheduling for the IBM RISC System/6000 processor," *IBM J. Res. Develop.*, vol. 34, pp. 85–92, 1990.

[6] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.*, vol. 11, pp. 25–33, 1967.

[7] V. Propescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The megaflow architecture," *IEEE Micro*, June 1991.

[8] J. R. Ellis, *Bulldog: A Compiler for VLIW Architecture*. Cambridge, MA: The M.I.T. Press, 1986.

[9] J. Bruno, J. W. Jones, and K. So, "Deterministic scheduling with pipelined processors," *IEEE Trans. Comput.*, vol. C-29, pp. 308–316, 1980.

[10] J. L. Hennessy and T. R. Gross, "Postpass code optimization of pipeline constraints," *ACM Trans. Programming Language and System*, vol. 5, pp. 442–448, 1983.

[11] J. A. Fisher, "The VLIW machine: A multiprocessor for compiling scientific code," *IEEE Comput.*, pp. 45–53, July 1984.

[12] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. C-30, pp. 478–490, 1981.

[13] S. Melvin, "Exploiting fine-grained parallelism through a combination of hardware and software techniques," in *Proc. Int. Conf. Parallel Comput.*, 1991.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completness*. San Francisco, CA: Freedman, 1979.

[15] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez Jr, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, pp. 137–146, 1972.

[16] C. I. Yang, J. S. Wang, and R. C. T. Lee, "A branch-and-bound algorithm to solve the equal-execution-time job scheduling problem with precedence constraint and profile," *Comput. Oper. Res.*, vol. 16, pp. 257–269, 1989.

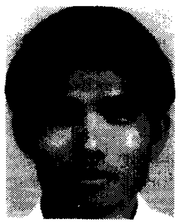
[17] H. C. Chou, "A study of superscalar instruction scheduling problem," Ph.D. dissertation, Inst. Comput. Sci. Inform. Eng., Nat. Chiao Tung Univ., Taiwan, 1992.

[18] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rhinehart, and Winston, 1976.

[19] M. R. Edward, N. N. Jurg, and D. Narsingh, *Combinatiorial Algorithms: Theory and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

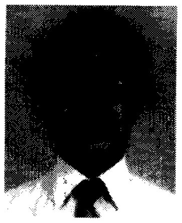
[20] Y. H. Shiau and C. P. Chung, "Effects and handling of instruction class contention in superscalar processing," submitted paper.

[21] E. Lawer, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer, "Pipeline scheduling: A survey," IBM Res. Rep. RJ 5738, San Jose, CA, 1987.



Hong-Chich Chou was born in Taiwan, Republic of China. He received the B.S., M.S., and Ph.D. degrees in computer science and information engineering from the National Chiao Tung University, Taiwan, in 1986, 1988 and 1992, respectively.

Currently, he is with the Computer & Communication Research Laboratories (CCL) of Industrial Technology Research Institute (ITRI) as a hardware engineer. His research interests include computer architecture, parallel processing and VLSI system design.



Chung-Ping Chung received the B.E. degree from the National Cheng Kung University, Taiwan, Republic of China, in 1976, and the M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in electrical engineering.

He was a Lecturer of electrical engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with the Department of Computer Science and Information Engineering at the National Chiao Tung University, Hsinchu, Taiwan, ROC, where he is a Professor. From 1991 to 1992, he was a Visiting Association Professor of Computer Science at the Michigan State University. His research interests include computer architecture, parallel processing, and parallel compiler design.