A preliminary version of this paper appeared in ICPP '94

Statement Re-ordering for DOACROSS Loops *

Ding-Kai Chen
dkchen@mti.sgi.com
Silicon Graphics
Computer Systems

Pen-Chung Yew
yew@cs.umn.edu
Dept. of Computer Science
University of Minnesota

April 13, 1995

Abstract

In this paper, we propose a new statement re-ordering algorithm for DOACROSS loops that overcomes some of the problems in the previous schemes. The new algorithm uses a hierarchical approach to locate strongly dependent statement groups and to order these groups considering critical dependences. A new optimization problem, dependence covering maximization, which was not discussed before is also introduced. It is shown that this optimization problem is NP-complete, and a heuristic algorithm is incorporated in our algorithm. Run-time complexity analysis is given for both algorithms. This new statement re-ordering scheme, combined with the dependence covering maximization, can be an important compiler optimization to parallelize loop structures for large scale coarse and fine grain parallelism.

Keywords: Compiler Optimization, Data Dependence, Doacross Execution, Redundant Synchronization Elimination, Statement Re-ordering.

*This work was supported in part by the National Science Foundation under Grant Nos. NSF MIP-8920891, NSF MIP-9307910, and the U.S. Department of Energy, Grant No. DOE DE-FG02-85ER25001.

Contents

1 Introduction

Loop-level parallelism is very important in achieving high performance on multiprocessors. Many architectures and optimization compilers are aimed at efficient execution of parallel loops. One kind of DO loop, the *DOALL* loops, do not have loop-carried dependences, and it is relatively easy to obtain the desired speedup. On the other hand, it is much harder to parallelize *DOACROSS* loops [15, 16, 7, 8], which contain loop-carried dependences.[1] They require the consideration of not only the scheduling and the load-balancing issues, but also the synchronization issues, which are usually much more difficult.

Loop-carried dependences can be categorized as *lexically forward* and *lexically backward*. Vector and SIMD machines can handle DOACROSS loops with only lexically forward dependences [1, 15, 22, 14]. One advantage of the MIMD (multiple-instruction-multiple-data) machines is it allows loops with backward loop-carried dependences to be handled by DOACROSS execution (i.e., by delaying consecutive iterations to satisfy backward dependences). The speedup obtained in this way can be large and therefore pay off the extra synchronization overhead.

Because the order of the statements in a loop determines the amount of delay between consecutive iterations, the extent of the overlap between iterations, or equivalently, the parallelism, is determined solely by the statement order. It is very important in DOACROSS loops to order the statements for the minimal delay (or the maximal overlap) between iterations. As has been proven by Cytron [7], the optimization problem of statement re-ordering is NP-complete. Hence, in practice, algorithms using heuristics have to be used.

Another related optimization problem, which has been overlooked in the past, is to reduce the amount of synchronization through statement re-ordering. By making more dependences covered by others, more synchronization becomes "redundant" and therefore can be eliminated without affecting the correctness of the execution [12, 10, 13, 6]. This problem should be considered along with the delay minimization problem mentioned above. Although there is a trade-off between parallelism and synchronization overhead, we shall make parallelism our main optimization goal. The reduced synchronization will be achieved without compromising the parallelism available.

This paper describes a compiler optimization strategy that performs statement re-ordering for DOACROSS loops to maximize the parallelism and to minimize the synchronization requirement. Unlike previous works, we adopt a hierarchical

[1] The definition of DOACROSS loops used here includes all the loops with loop-carried dependences. These loops include those classified by the traditional DOACROSS loop definition in which only loops with dependence cycles are considered DOACROSS loops.

approach to find closely connected components in the dependence graph and to schedule statements in each component together. The synchronization reduction, which was not considered in the previous works, is performed when statements can be re-ordered without affecting overall parallelism. We organize our presentation as follows. Section 2 gives background and definitions for later discussions. In Section 3, the proposed hierarchical re-ordering strategy is explained in detail. Preliminary performance results and discussions are presented in Sections 4 and 5. Section 6 concludes our presentation.

2 Background

2.1 Dependences

The essential requirement for a correct execution of a program in parallel is to preserve dependences implied by the program semantics. There are two kinds of dependences, *data dependences* and *control dependences*.

A *data dependence* occurs when a data object x is accessed by two statements s_p and s_q, and there exists an execution path between s_p and s_q. Also, at least one of the accesses is a write access. Another kind of dependences, *control dependences*, occur when the result of an expression alters the course of execution, and later statements are said to be "control-dependent" on the expression. There are techniques that transform cross-iteration control dependences into data dependences [2]. Therefore, in this paper we focus only on data dependences.

Dependences can also be classified as *loop-independent* or *loop-carried*. Basically, loop-independent dependences occur between statement instances within one iteration while loop-carried dependences occur between statement instances of different iterations. Alternatively, for a single loop, loop-independent dependences have source and sink statements in the same iteration with a zero dependence distance; loop-carried dependences have a (> 0) dependence distance.

Furthermore, a dependence can be described as either (lexically) *forward* or *backward*. Given a statement order in a loop, with the first statement at the top and the last statement at the bottom of the loop, a forward dependence stems from a statement closer to the top to a statement closer to the bottom of the loop. Similarly, a dependence with the location of the sink statement being closer to the top than, or at the same statement as, the source statement is a backward dependence.

Given the above descriptions, we have the following observations which are essential to our strategies.

Observations

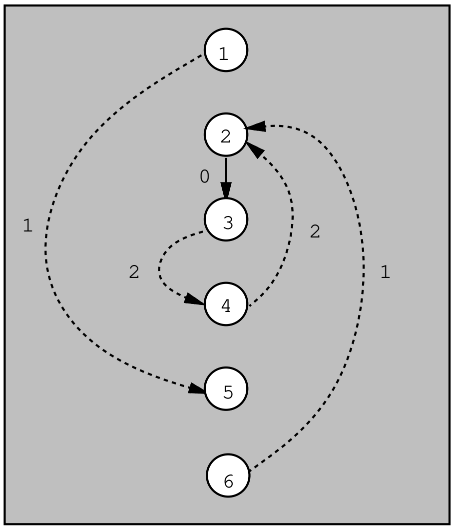
- A backward dependence *must* be a loop-carried dependence.
- A forward dependence can be *either* loop-carried *or* loop-independent. But, a loop-independent dependence *must* be a forward dependence.
- The order of the source and the sink statements of a loop-independent dependence has to be kept when we re-order the statements. The order of the source and the sink statements of a loop-carried dependence can be reversed, and the dependence will not be violated.
- The properties of being "forward" and "backward" are defined only when a statement order is specified, whereas the properties of being "loop-carried" and "loop-independent" are independent of the statement order.

Finally, a *dependence graph* is a directed graph describing dependence relations among statements. Each node (vertex) is a statement and each edge (arc) represents a dependence. The edges can be annotated with information such as the type of dependence, and the dependence distance. Note that there can be multiple edges between two nodes if there is more than one dependence with different dependence distances. A *strongly connected component* (SCC) in a directed graph is a maximal subgraph such that each node in this subgraph has a directed path to any other node in the same subgraph [18].

2.2 DOACROSS Execution

DOACROSS execution for the loops with loop-carried dependences is achieved by requiring the instances of the sink statements in later iterations to wait for the completion of the source statements in earlier iterations, usually with explicit synchronization between source and sink statements. Because the dependences derived from the sequential execution never form a cycle among all statement instances, such synchronization will not cause dead-locks during execution. The key to the DOACROSS execution is the *delay* of the consecutive iterations when there is any backward dependence. For example, Figure 1a shows a DOACROSS loop, with its dependence graph shown in Figure 1b. There are two backward dependences ($4 \xrightarrow{2} 2$), and ($6 \xrightarrow{1} 2$), where the numbers associated with the "$\longrightarrow$" show the dependence distances. The second backward dependence causes a delay of 5 time units for each iteration, as seen in Figure 1c. There are also three forward dependences: ($2 \xrightarrow{0} 3$), ($1 \xrightarrow{1} 5$), and ($3 \xrightarrow{2} 4$), where only the first one is loop-independent. In addition, although there are a total of 4 loop-carried dependences, synchronization for ($6 \xrightarrow{1} 2$) will preserve all other loop-carried dependences.

```
DO I=0,N−1
s1:   A[I+1]=...
s2:   X      =C[I]+D[I]
s3:   B[I]   =X
s4:   C[I+2]=B[I−2]
s5:   ...    =A[I]
s6:   D[I+1]=...
END DO
```
(a)



(b)

Iterations	1	2	3
time 1	s1		
time 2	s2		
time 3	s3		
time 4	s4		
time 5	s5		
time 6	s6	s1	
time 7		s2	
time 8		s3	
time 9		s4	
time 10		s5	
time 11		s6	s1
time 12			s2
time 13			s3
time 14			s4
time 15			s5
time 16			s6
time 17			

(c)

Figure 1: DOACROSS loop example.

Therefore, only the explicit synchronization between statements 2 and 6 is necessary. All other synchronization becomes "redundant." It is important to note that the delay caused by the backward dependences determines the overlap between iterations. Explicit synchronization is necessary to effect such a delay in a MIMD machine. On the other hand, the delay can be used to estimate when the dependences are most likely to be satisfied and to reduce unnecessary busy waiting at the sink statements.

Apparently, parallelism obtained in Figure 1 is very small due to very little execution overlap between iterations. As mentioned in the previous section, as long as the order specified by any loop-carried dependence is not violated, we can re-order the statement to minimize the delay and to increase the overlap and hence the parallelism. Given a statement order, a backward loop-carried dependence ($src \xrightarrow{d} sink$) requires a delay of

$$\frac{src - sink + 1}{d}. \qquad (1)$$

We call $(src - sink + 1)$ the *statement distance* of the dependence. It is the longest delay among all backward dependences that determines the final delay. Figure 2a shows a much better statement order for the example in Figure 1a which requires a delay of $\frac{6-5+1}{2} = \frac{5-4+1}{2} = 1$. It can be seen that one of the forward dependences ($3 \xrightarrow{2} 4$) has been converted to a backward dependence, and one of the backward dependences ($6 \xrightarrow{1} 2$) becomes a forward dependence in this new

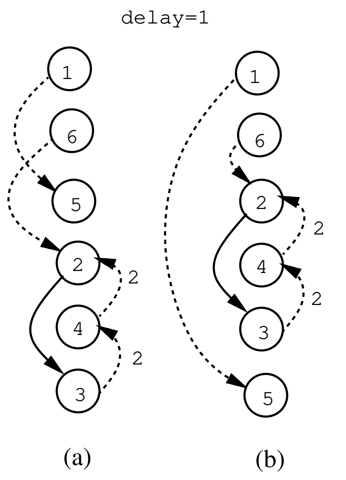


Figure 2: Optimized statement orders.

order, but the order between s_2 and s_3, which is implied by the only loop-independent dependence, is not changed. More parallelism usually requires more communication overhead. In our new statement order, $(s_1 \xrightarrow{1} s_5)$ and $(s_6 \xrightarrow{1} s_2)$ have to be explicitly synchronized because no other synchronization covers them.

Given the same theoretical delay value, we want to have as much dependences covered as possible and therefore minimize the synchronization requirement. Figure 2b shows a slightly different statement order than that of Figure 2a. It has the same delay of 1 but the synchronization for $(s_1 \xrightarrow{1} s_5)$ is now covered by that of $(s_6 \xrightarrow{1} s_2)$ because we can follow the order $(s_1 \xrightarrow{0} s_6 \xrightarrow{1} s_2 \xrightarrow{0} s_5)$ to ensure the order of $(s_1 \xrightarrow{1} s_5)$. Therefore, the synchronization for $(s_1 \xrightarrow{1} s_5)$ is redundant and unnecessary.

The optimization problem discussed so far can be formally stated as follows.

Problem Statement

Given a dependence graph $G(V, E)$, where $V = \{s_i\}$ is the set of statements in a loop and $E = \{(src_i \xrightarrow{dist_i} sink_i)\}$ is the set of dependences among members of V, we want to find a mapping $P : V \rightarrow \{1, \ldots, |V|\}$ such that

1. *$delay = \max_{i \in E} \frac{src_i - sink_i + 1}{dist_i}$ is minimized, and*

2. *number of redundant synchronization is maximized.*

Obviously the first goal must have a higher priority; otherwise, a serial execution of the loop requires no synchronization, but it has very poor performance.

We assume execution in each iteration is sequential and each statement takes one unit time to execute. Although the scheme described here is easier to understand for single loops, or the innermost loops, it can be extended to multiply-nested loops and is discussed in Section 5.

3 Re-ordering Strategies

3.1 Previous Works

It has been shown that even for a simpler dependence graph with only loop-carried dependences (hence, no restriction on its statement ordering), the delay minimization problem is NP-complete [7].

The heuristics algorithms previously proposed basically have two phases. In the first phase, the graph is partitioned and the partitions are then ordered. In the second phase, statements within each partition are ordered. The idea is to determine the partition order in the first phase such that, during the local optimization in the second phase, no loop-independent dependences will be violated.

In the method proposed by Cytron [7] in the partitioning phase, it forms antichain-rows by level-sorting the dependence graph considering only forward dependences. The source and the sink of a forward dependence are placed in different partitions. In the second phase, a *weight* function for each statement s_i in a partition (or an antichain-row) is calculated. If the maximal statement distance among all the backward dependences in which s_i is the sink is p, and the maximal statement distance among all the backward dependences in which s_i is the source is q, the weight of s_i is $p - q$. The statements within each partition are then re-ordered according to their weight (in an increasing order). Figure 3a shows the antichain-row partitioning after the first phase and the corresponding weight for each statement. A final order is shown in Figure 3b with a delay of 2.

From the observations in Section 2.1, we know that the statement order of a forward dependence need not be preserved, unless it is a loop-independent dependence. That means the partitioning phase of Cytron's method is overly conservative. For example, interchanging the positions of s_3 and s_4 results in a smaller delay. Moreover, the property of "forward-ness" and the weight function depend on a given statement order, and the optimization is likely to reach only a local optimum. In fact, anomalies could occur where the "optimized" order actually has a longer delay.

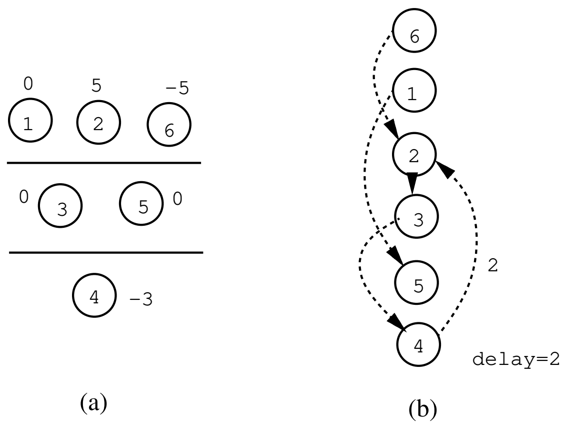


Figure 3: Cytron's algorithm.

A similar method was also proposed by Simons et. al. [3, 19]. In their first phase, statements are level-sorted into partitions considering only the loop-independent dependences. The source and the sink statements of a loop-independent dependences are placed in different partitions. Within each partition, further partitioning is performed repeatedly until no source and sink statements of a loop-carried dependence are contained in the same partition. Given an integer k, which is the desired bound of the delay, their algorithm tries to find an order that orders statements within each partition such that the statement distance of any backward dependence (now crossing partition boundaries) is less than or equal to k. For some special backward dependences such as multiple chains and out- or in-trees, this algorithm finds an order for a given k if one exists. However, sometimes the longer delay can result from the fact that the partition bound the movement of statements in the second phase, unfortunately, a wrong decision on the direction of loop-carried dependences is made in the initial partitioning.

In summary, the two-phase algorithms try to preserve loop-independent dependences in the first phase without a proper consideration of all the loop-carried dependences, which limits the strategies that can be used in the second phase.

3.2 Hierarchical Scheduling – A New Approach

Because it is not adequate to preserve all forward dependences [7], and it is difficult to determine which loop-carried dependences should become forward dependences in the first phase [3], our strategy is to postpone making the decision for each loop-carried dependence until it is necessary. Another intuition behind our strategy is that if we can make

Input: *a dependence graph from a DOACROSS loop.*
Output: *a statement order with minimized delay and maximized dependence covering.*
Algorithm *HS*
begin
    normalize the dependence graph;
    find SCCs;
    if *the total number of SCCs>1* **then**
        level-sort SCCs;
        order SCCs in each level and maximize the dependence covering; // see Section 3.3
    endif;
    foreach *SCC s* **do**
        PSCC_scheduling(s);
    endfor;
end;

procedure *PSCC_scheduling(S)*
begin
    find PSCCs in S; // see Section 3.2.1
    order PSCCs to minimize delay; // see Section 3.2.2
    foreach *PSCC s* **do**
        PSCC_scheduling(s);
    endfor;
end;

Figure 4: Hierarchical scheduling algorithm outlines.

the source and the sink statements of a dependence close to each other, even if the dependence becomes backward, the resulting delay would still be small. Conceptually, our proposed *Hierarchical Scheduling* (HS) strategy finds *Pseudo-Strongly Connected Components* (PSCC) first and orders the components to minimize the potential statement distances of the backward dependences among the components. For each PSCC, this process is recursively applied until all statements are scheduled. Figure 4 outlines the major steps of the proposed HS algorithm. The algorithm is explained in detail in later sections.

Basically, the algorithm recognizes the fact that statements that are strongly connected in a dependence graph should be scheduled close to each other in order to minimize statement distance of any backward dependence. The dependence graph is first normalized such that if there is more than one dependence between the same source and sink statements, we keep only the one with the smallest dependence distance. Next, we find all of the strongly connected components (SCCs) in the dependence graph. They are also called π-*blocks* in [11, 15]. After they are ordered to maximize the dependence covering, statements within each SCC are then ordered. Using the same heuristics, we try to determine, in each SCC, the strongly connected sub-structures (PSCCs), which consist of closely related statements in terms of their dependences. They are ordered and the process proceeds recursively.

3.2.1 PSCC Identification

PSCCs are strongly connected subgraphs within a strongly connected graph such that they partition the original graph. In the main algorithm, we use Tarjan's [21] well-known algorithm to determine SCCs. However, given a strongly connected graph G, we cannot use the same algorithm to find PSCC because we will obtain the same graph as G. If G has n nodes, the first PSCC is determined by identifying the largest connected subgraph that has less than n nodes. This is achieved by iteratively selecting a node v to be excluded from the PSCC and then using Tarjan's algorithm to find SCCs in $G - \{v\}$, which always have $n - 1$ nodes. The largest SCC found in this process is the first PSCC. After the first PSCC is found, it is deleted from G, and Tarjan's algorithm is used to find the remaining PSCCs.

It is not difficult to see that the first PSCC found by the above process is the largest connected subgraph. If we condense the input graph G such that each PSCC becomes a node and we keep only the edge with the smallest dependence distance between the nodes, the resulting graph G' will be a ring.

Lemma 1 *Each node in G' representing a PSCC has exactly one predecessor and one successor.*

Proof: See Appendix. □

Theorem 1 *G' is a ring.*

Proof: The proof follows Lemma 1 and the fact that G' is strongly connected. □

3.2.2 PSCC Ordering

From Theorem 1, we can see that the PSCCs form a ring. The goal of ordering the PSCC is to minimize the distance between the source and the sink PSCCs of any backward dependences under the constraint that no loop-independent dependence (with dependence distance 0) can go backward. There are three cases to consider:

Case 1: If there is no loop-independent dependence in the ring of PSCCs, we select one forward edge e by the following criteria:

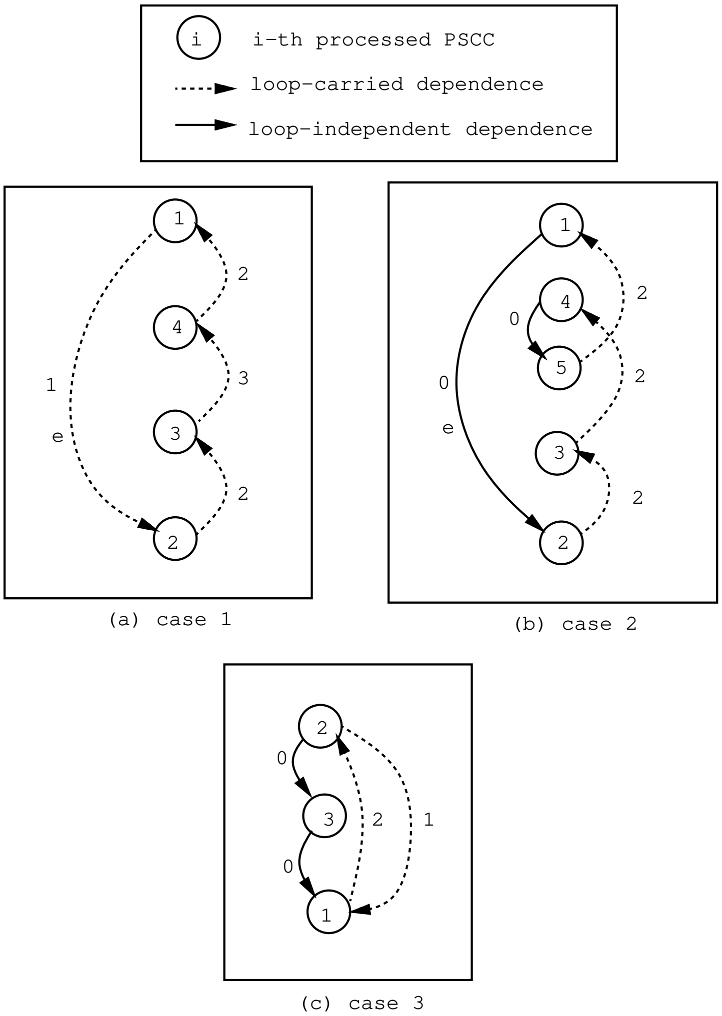


Figure 5: SCC ordering examples.

1. it has the smallest dependence distance, and
2. in case there is a tie, we select the edge with the largest total number of statements in its source and sink PSCCs.

We make the source of e the first PSCC and its sink the last PSCC. All other edges are backward edges from a PSCC to a PSCC immediately above it (see Figure 5a). The rationale behind this is to minimize the statement distance between source and sink PSCCs.

Case 2: If some of the edges correspond to loop-independent dependences, their order has to be preserved. Similar to the previous case, we select one forward edge e with one more constraint that the out edge of e's sink node $pscc_{last}$ has a distance other than 0. Such an edge e must exist because, according to our assumption, not all the edges are with distance 0. Edge e will have distance 0 in this case. We then start from e's sink $pscc_{last}$ to construct an upward path (see Figure 5b). For each node $pscc_p$ encountered, if $pscc_p$'s out edge e' is a loop-carried dependence, its sink node $pscc_q$ is temporarily put right below the source of e. Otherwise, if e' is a loop-independent dependence, its sink node $pscc_q$ is put right below the current node $pscc_p$. The rationale is similar to the above except that the order of the loop-independent dependences have to be preserved. Figure 5b shows the ordering of a 5-node ring with both loop-carried and loop-independent dependences.

Case 3: In rare occasions, all edges between PSCCs have distance 0. Figure 5c gives an example in which nodes 1 and 2 form a PSCC and node 3 itself forms another PSCC and the two edges between these two PSCCs are all loop-independent dependences. To handle these exceptional cases, we go back to the strongly connected input dependence G and select a node v whose minimal dependence distance among its outgoing edges is the largest of all nodes. All outgoing edges of node v must be loop-carried. Otherwise it implies that every node is the source of some loop-independent dependences, and we will find a cycle of loop-independent dependences, which is not possible. Node v will be the last statement of all statements in G and is deleted from G. The SCCs are determined in $G - \{v\}$ and are level-sorted. The scheduling process is then resumed.

After all PSCCs are scheduled, each PSCC is decomposed and the sub-structures ordered recursively until the leaf PSCCs containing a single node are reached.

3.3 Dependence Covering Maximization

So far, our ordering strategy has been focused primarily on minimizing the delay (i.e., maximizing the parallelism). We should also considering reducing the needed explicit synchronization by maximizing the dependence covering, if possible. It has to be done without compromising the parallelism that is our the ultimate goal. Maximizing dependence covering is best considered in the main algorithm, after the SCCs are determined and level-sorted. Because SCCs in each level are independent of each other, the new order will not affect the delay and, hence, the parallelism. On the other hand, dependence covering can be enhanced if the SCCs in the same level are ordered carefully. Figure 6a shows the four SCCs found for the example of Figure 1. Individually, both dependences $scc_a \rightarrow scc_c$ and $scc_b \rightarrow scc_d$ require synchronization. If they are ordered as $(scc_a, scc_b, scc_d, scc_c)$ or $(scc_b, scc_a, scc_c, scc_d)$ (see Figure 6b), one of the dependences could be covered.[2] On the other hand, if they are ordered as shown in Figure 6c, no dependence can be covered. Hence, the key here is to avoid the dependence edge crossings as much as we could.

Our strategy, therefore, is to use a greedy algorithm to minimize such edge crossings when ordering the SCCs in a level. We schedules one level at a time starting from the topmost level. In each level, a *Source Vector* is generated for each SCC. The source vector of a SCC scc_p is a list of integers, ordered left to right from small to large, which correspond to the sequence number of the SCCs in the previous levels whose dependence sink is scc_p. For example, suppose after the scheduling of the first level, scc_a and scc_b are ordered as (scc_a, scc_b). The source vectors for scc_c and scc_d will be (1) and (2), respectively.

Given an order of the m SCCs in the current level, an *Ordered Source Vector* $OSV = (sv_m, \ldots, sv_1) = (osv_1, \ldots, osv_n)$, where n is the sum of the vector length of the m source vectors, can be constructed from the source vectors by placing them left to right with the first SCC's vector being the rightmost one. An *inversion* in OSV occurs when we have $osv_i > osv_j$ while $i < j$.

Theorem 2 *The number of inversions that occur in the ordered source vector is equal to the number of edge crossings.*

Proof: We shall prove by induction on the number of edges n.

Base case, n=1: When there is only one edge, there is no edge crossing. On the other hand, there is only one component in the ordered source vector; hence no inversion. Suppose the theorem is true for $n = k$. We need to show that the theorem is

[2] We need to construct the control path graph (CPG) to determine whether the dependence covering does occur [6].

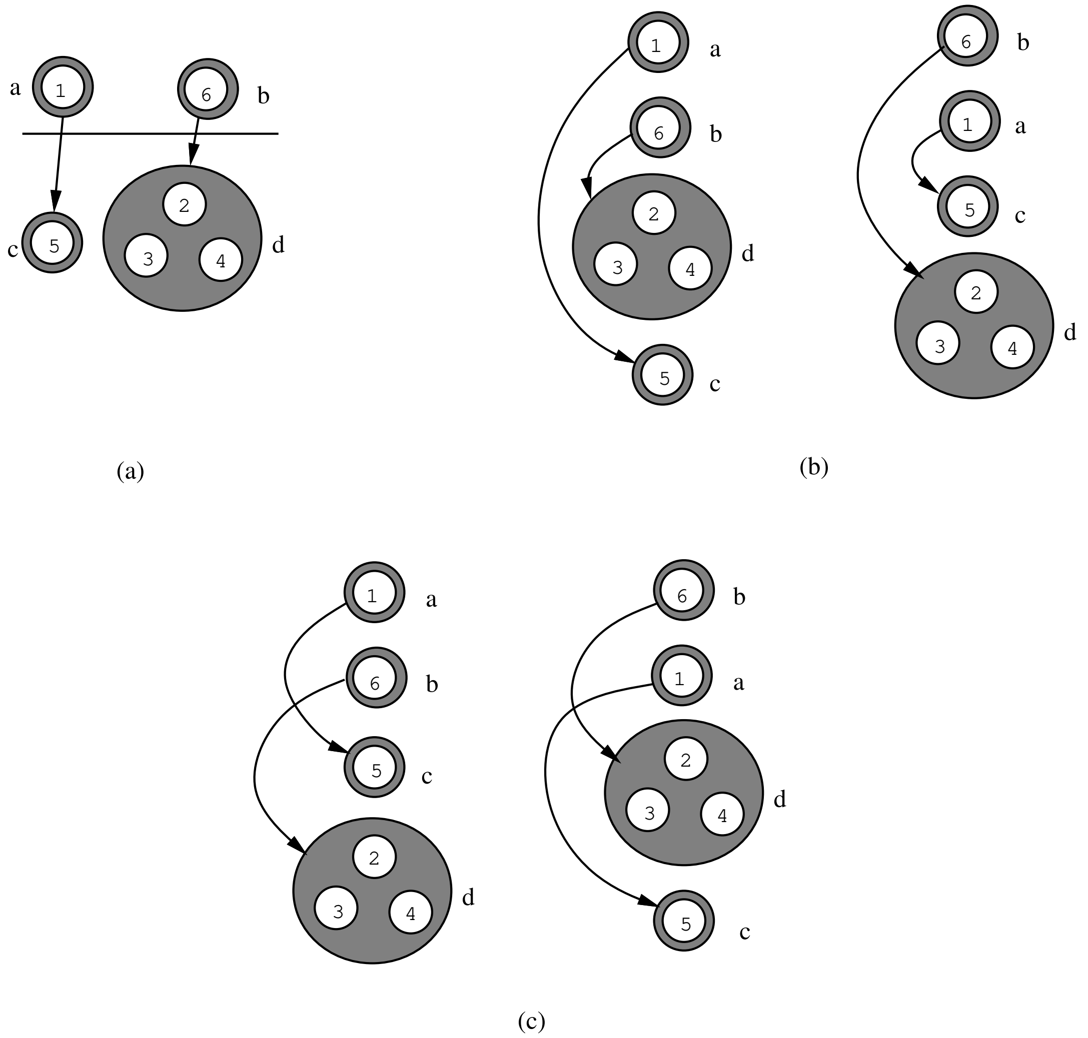


Figure 6: Dependence covering examples.

also true for $n = k + 1$. We select an edge e from the dependence graph whose source SCC src has the smallest sequence number. If there is a tie, the one whose sink SCC $sink$ has the largest sequence number is chosen. If we remove this edge and its corresponding component in the ordered source vector, we have k edges left and from the induction hypothesis, the number of inversions in the ordered source vector equals the number of edge crossings. When we re-install the removed edge and its corresponding component osv_e to the ordered source vector, the new edge crossings must be from SCCs with a larger sequence number than that of src to SCCs with a larger sequence number than that of $sink$. It implies that, in the ordered source vector, the components of the edges intersecting e must be placed to the left of where osv_e is placed and have values greater than osv_e. Therefore, the additional number of inversions involving osv_e is exactly the number of new edge crossings caused by putting e back, and the theorem holds for $n = k + 1$. □

With the number of inversions in the ordered source vector, we can estimate the potential for dependence covering. A smaller number of inversions implies higher covering potential. Our optimization problem can thus be reduced to finding an order such that the number of inversions in the ordered source vector is minimized. This is equivalent to solving the following special case of the *QUADRATIC ASSIGNMENT PROBLEM* [9] with the $cost_{ij}$ being the number of inversions in (sv_i, sv_j) which is constructed from the source vectors of SCC i and j:

Problem: Assume we have non-negative integer costs $cost_{ij}$, $1 \leq i, j \leq n$ and distances d_{kl}, $1 \leq k, l \leq n$, where

$$d_{kl} = \begin{cases} 1 & if\ k < l \\ 0 & otherwise. \end{cases}$$

We want to find out whether there is a one-to-one function $f : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ such that

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} cost_{ij}\, d_{f(i)f(j)} \leq K$$

where K is a positive integer. Unfortunately, this problem is also NP-complete, as shown by Theorem 3 in the Appendix.[3] We use a heuristic algorithm shown in Figure 7 to determine the best SCC order.

[3] Actually this is only a subproblem because in addition to minimizing the edge crossings caused by the SCCs above the current level, we should also minimize the edge crossings caused by the SCCs below the current level. This harder problem is also NP-complete.

Input*: a set of SCCs and a cost matrix which gives number of edge crossings of each ordered SCC pair* (i, j).
Output*: an order of SCCs with minimized dependence edge crossings.*
Algorithm *DC*
begin
    foreach *ordered SCC pair* (i, j) **do**
        if $cost(i, j) < cost(j, i)$ **then**
            add ('i $\rightarrow$ *j',* $cost(j, i) - cost(i, j)$*) to the edge list;*
        endif*;*
    endfor*;*
    sort the edge list according to the second components;
    foreach *the edges in the edge list* **do**
        get the ordered SCC pair (i, j) *with the next largest second component;*
        if *there is no path from* j *to* i **then**
            add the edge ('i $\rightarrow$ *j') between* i *and* j*;*
    endfor*;*
    topological sort SCCs to get the final order;
end*;*

Figure 7: SCC ordering algorithm to maximize dependence covering.

.

3.4 Run-time Complexity

Suppose there are $\mid V \mid$ statments and $\mid E \mid$ dependences. The normalization step of the Algorithm *HS* in Figure 4 examines dependence edges one at a time, so the time complexity is $O(\mid E \mid)$. Using Tarjan's algorithm to find SCCs takes $O(\mid E \mid + \mid V \mid)$ time steps and level-sorting takes $O(\mid E \mid + \mid V \mid)$ time steps. Procedure $PSCC\_scheduling()$ (see Figure 4) will not be called more than $\mid V \mid$ times. In each time, it needs $O(\mid V \mid \times (\mid E \mid + \mid V \mid))$ time steps to find PSCCs and $(\mid V \mid)$ time steps for ordering PSCCs if cases 1 and 2 are assumed. If in PSCC ordering, there is no loop-carried dependences among PSCC as in case 3, it requires $O(\mid E \mid + \mid V \mid)$ time steps to find out the last node v and other PSCCs, and to perform an additional level-sorting.

For dependence covering maximization, all dependence edges are examined at most once to form the source vectors of SCCs. It takes $O(\mid E \mid + \mid V \mid)$ time steps. Each source vector has at most $O(\mid V \mid)$ components and it takes $O(\mid V \mid \log \mid V \mid)$ time steps to sort the components. The inversions between each pair of SCCs can be determined in $O(\mid V \mid)$ time steps if the source vectors' components are sorted. The cost matrix has $O(\mid V \mid^2)$ entries and is available with $O(\mid V \mid^3)$ time steps. The dependence edge list is $(\mid V \mid^2)$ long and needs $O(\mid V \mid^2 \log \mid V \mid)$ time steps to sort. To process the edge list and maintain the reaching information takes at most $O(\mid V \mid^4)$ time steps.

Routine Name	Line Number	# Nodes	# Edges	Percentage Parallelism (%)				Parallelism
				ORG	CYT	SMA	NEW	(1000 iter)
BISECT	118	53	103	9.43	11.32	41.51	83.02	5.86
BISECT	183	43	55	20.93	20.93	58.14	72.09	3.57
BISECT	221	21	19	76.19	90.48	85.71	90.48	10.40
CHWZR	66	24	19	91.67	91.67	91.67	91.67	11.87
CHWZR	80	70	68	95.71	94.29	92.86	95.71	22.82
HTRIB3	72	70	68	95.71	91.43	91.43	95.71	22.82
HTRIDI	77	67	67	95.52	95.52	94.03	95.52	21.87
IMTQL1	50	11	8	54.55	81.82	81.82	81.82	5.48
IMTQL1	78	60	132	33.33	50.00	20.00	71.67	3.52
TINVIT	125	113	221	18.58	15.04	32.74	88.50	8.63
TINVIT	155	32	36	53.12	31.25	37.50	75.00	3.99

Table 1: Percentage parallelism for several inner loops from Eispack.

To summarize, the overall time complexity without dependence covering maximization is $O(|V|^2 \times (|E| + |V|))$, and is $O(|V|^4)$ with dependence covering maximization.

4 Performance Results

To see how effective the proposed hierarchical scheduling strategy is, we test it on several dependence graphs. These graphs are all from the inner loops of several subroutines of the Eispack (a set of mathematics library routines developed at Argonne National Laboratory to solve eigenvalues and eigenvectors [20]). We randomly chose one test driver routine `chtest.f` and used a source-level performance analysis tool, MaxPar [4, 5], to obtain loop-carried flow dependence information.[4] A dependence graph from each inner loop was constructed from its intermediate form by an experimental optimizing compiler developed under EPG-sim environment [17] and was augmented with the loop-carried dependence information mentioned above. Finally, the dependence graphs were used as inputs to the two algorithms described in Section 3.1 and to the new hierarchical scheduling algorithm.

The preliminary results for improved parallelism are shown in Table 1. The first two columns give the routine name and the location of the inner loops. The third and the fourth columns described their dependence graphs. The next four columns

[4] We assume the anti- and output-dependences can be eliminated by the optimization using scalar and array expansion or privatization.

show the *Percentage Parallelism*, which is defined as,

$$\frac{number\ of\ nodes - delay}{number\ of\ nodes}$$

for the original loops and for the loops optimized by different algorithms. The best percentage parallelism is 100%, and a 0% parallelism indicates a serialized execution. Note that the results for Cytron's algorithm (shown under "CYT" column) are chosen from the best of three consecutive runs because each optimized result depends on its input statement order. An absolute parallelism assuming 1000 iterations is shown in the last column using the following formula

$$\frac{number\ of\ nodes \times 1000}{number\ of\ nodes + delay \times 999}.$$

This parallelism is a performance upper bound because

1. We assumed an unlimited amount of resource and no communication overhead.
2. The loop-carried dependence information is obtained at run-time, which is less conservative than at compile time.
3. The intermediate form is not optimized; therefore it tends to have more nodes than an optimized one.

It can be seen that the performance of the proposed algorithm (shown under the "NEW" column) is consistently better than the previous schemes. In both Cytron's and Simon's schemes (shown under the "CYT" and the "SMA" columns), there are loops whose original delay is shorter than the optimized delay. It is also important to note that the small difference in the percentage parallelism is actually quite significant when it approaches 100%. It is because if we replace the delay by $(1 - percentage\ parallelism) \times number\ of\ nodes$, then the absolute parallelism can be approximated by

$$\frac{1}{1 - percentage\ parallelism} = \frac{number\ of\ nodes}{delay}.$$

These performance results confirm our expectation that the hierarchical scheduling strategy can achieve a higher performance than previous schemes. However, we still need more experiments for dependence covering maximization and using compile time available dependence information.

5 Discussions

5.1 Dependence Distance

One of the advantages of our scheme is to take the dependence distance into consideration in statement re-ordering. Previous algorithms either ignore the dependence distances or unroll the loops to reduce the largest distance to one. Ignoring dependence distances will treat all dependences equally and will fail to identify those that should stay forward to avoid large delays. Unrolling the loop can create a lot of loop-independent dependences in the unrolled loop which can further restrict a possible optimal ordering.

5.2 Multiply-nested Loops

An immediate question to the proposed re-ordering algorithm is whether it can handle dependence graphs from multiply-nested loops. In such cases, the dependence distance vectors will have more than one component (instead of the scalar d used in Equation (1) in page 4). It is shown that to determine the optimal delays for each loop level requires using the *simplex algorithm* to solve a linear programming problem [7]. Our problem is slightly different in that we try to determine an order to minimize the delays. Therefore, the goal is to minimize the statement distance for *critical* backward loop-carried dependences. They are usually indicated by small dependence distance vectors. The identification of critical dependences is needed only in the PSCC ordering step (see Figure 4). For example, in the case 1 of the PSCC ordering, we want the dependence with smaller dependence distance vector to be the forward dependence. One way to determine the critical dependence is to compare the sum of the components of the distance vectors.

For non-perfectly nested loops, statements in the innermost perfectly-nested loops should be re-ordered first. The perfectly-nested loops are then considered as a whole as a “compound statement.” The dependence graph is modified accordingly. The re-ordering process can then proceed to process the new innermost perfectly-nested loops.

5.3 Implementation Issues

There are other improvements worth consideration. For example, when the total number of statements becomes small as the recursive decomposition proceeds, we could apply different methods, such as those proposed previously or even a branch-and-bound algorithm, and select the best final order. When ordering a ring of PSCCs without loop-independent

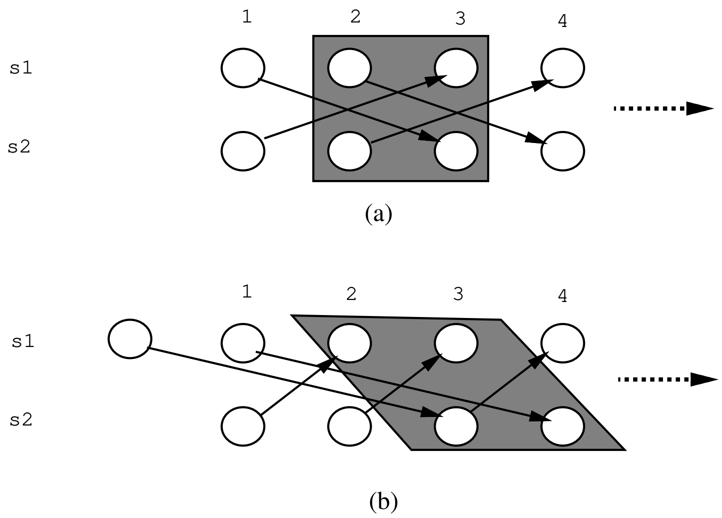


Figure 8: Loop alignment examples.

dependences in the ring (see case 1 in Section 3.2.2), we can form more than one forward dependence, instead of just one, if it is beneficial, because the increase in the statement distance for the remaining backward dependences is still small while the originally critical backward dependences (the ones that define the delay) are converted to forward dependences.

In dependence covering maximization (see Section 3.3), the source vectors of each SCC is calculated from the original dependence graph. However, when ordering the SCCs of the level i, some of the dependences have been covered because the order of SCCs in the previous levels has been determined and these dependences should not be used to determine the inversions in the current level i.

5.4 Future Works

Several other loop transformations can potentially improve the performance of the statement re-ordering. One of them is loop alignment [15], which changes the distance of the dependences. Figure 8 gives a loop alignment example. The original dependences among statement instances are shown in Figure 8a, while the dependences of the aligned loop are in Figure 8b. In the original loop, no statement re-ordering can improve the loop-level parallelism. After the alignment, the dependence distances become 1 and 3, and we can re-order the statement to make the backward loop-carried dependence have a larger distance and the delay is reduced from $\frac{2}{2}$ to $\frac{2}{3}$.

The above transformation can be seen as a way to move intra-loop parallelism to inter-loop parallelism because the independent statements (in shaded area) were in 2 iterations but are in 3 iterations after alignment. In future multiprocessors, each processor can exploit instruction-level parallelism in addition to the loop-level parallelism. Loop unrolling has been used extensively to improve intra-loop parallelism for a uniprocessor system. The balance between the two kinds of parallelism using techniques such as loop alignment and unrolling when communication overhead is considered will be an interesting research topic.

6 Conclusions

In this paper, we described a new statement re-ordering algorithm for DOACROSS loops. The new algorithm uses a hierarchical approach to locate strongly dependent statement groups and to order these groups considering critical dependences. A new optimization problem, dependence covering maximization, which was never discussed, is introduced. It is shown that this optimization problem is NP-complete, and a heuristic algorithm is incorporated in the re-ordering algorithm. Comparison to previous algorithms and a run-time complexity analysis are also presented. This new statement re-ordering scheme, combined with the dependence covering maximization, can be an important compiler optimization to parallelize loop structures for both coarse- and fine-grain parallelism.

A Related proofs

Lemma 1 *Each node in G' representing a PSCC has exactly one predecessor and one successor.*

Proof: It is obvious that each node has at least one predecessor and one successor; otherwise the input graph G would not be strongly connected. The corresponding PSCC of a node can be (1) the first PSCC ($pscc_1$), or (2) those found after the first one is determined. These two cases are considered below.

Case 1: Suppose the corresponding PSCC is $pscc_1$ and hence is the largest connected subgraph. Assume that it has ≥ 2 distinct successors and two of them are $succ_1$, $succ_2$, and that one of its predecessors is $pred$. If $succ_1$ equals $pred$, then $pscc_1$ is not the largest because the nodes in $pscc_1$, $succ_1$ and $pred$ can form an even larger connected subgraph with number of nodes $\leq n-1$. Therefore neither $succ_1$ nor $succ_2$ is $pred$. Then there exists a path P between $succ_1$ and $pred$ which does not include $succ_2$. If no such path exists, we will consider a similar path between $succ_2$ and $pred$, instead. Because the nodes in $pscc_1$, and P can form an even larger connected subgraph with number of nodes $\leq n-1$, the existence of P implies that $pscc_1$ is not the largest one, a contradiction. The same argument can be used to show that $pscc_1$ has at most one predecessor.

Case 2: Suppose the corresponding PSCC is not $pscc_1$; hence it is an SCC (scc_1) in $G - PSCC_1$. Assume that it has ≥ 2 distinct successors and two of them are $succ_1$, $succ_2$, and one of its predecessor is $pred$. In G', we find a cycle C from scc_1 to $pred$ through $succ_1$ then to scc_i which does not include $succ_2$. If no such cycle exists, we will consider instead a similar cycle through $succ_2$. The existence of such a cycle is guaranteed because G is strongly connected. If C include the node of $pscc_1$, then $pscc_1$ is not the largest. On the other hand, if $pscc_1$ is not in C, then C is in $G - PSCC_1$ and is strongly connected, which implies that scc_i is not a SCC because it is not maximal, a contradiction. The same argument can be used to show that scc_i has at most one predecessor.

From the above argument, we conclude that each node in G' has exactly one predecessor and one successor. □

Theorem 3 *The dependence covering maximization problem is NP-complete.*

Proof: The dependence covering maximization problem tries to minimize the inversions in the ordered source vector, which is equivalent to solving the following subproblem of the *QUADRATIC ASSIGNMENT PROBLEM* [9] with the $cost_{ij}$ being the number of inversions in (sv_i, sv_j) where sv_i and sv_j are source vectors of SCC i and j:

INSTANCE: Non-negative integer costs $cost_{ij}$, $1 \leq i, j \leq n$ and distance d_{kl},

$$d_{kl} = \begin{cases} 1 & if\ k < l \\ 0 & otherwise. \end{cases}$$

for $1 \leq k, l \leq n$, bound $K \in Z^+$.

QUESTION: Is there a one-to-one function $f : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ such that

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} cost_{ij} d_{f(i)f(j)} \leq K \ ? \tag{2}$$

It is easy to see that the problem is in NP because a nondeterministic algorithm need only guess an order of $\{1, 2, \ldots, n\}$ and check in polynomial time whether the total number of inversions is $\leq K$. We transform the following *FEEDBACK ARC SET* problem [9] to our quadratic assignment problem:

INSTANCE: Directed graph $G = (V, A)$, positive integer $K \leq\mid A \mid$.

QUESTION: Is there a subset $A' \subseteq A$ with $\mid A' \mid\leq K$ such that A' contains at least one arc from every directed cycle in G?

The transformation simply let

$$cost_{ij} = \begin{cases} 1 & if\ (i, j) \in A \\ 0 & otherwise. \end{cases}$$

We need to show that "the subset A' exists" $\Longleftrightarrow$ "Equation (2) can be satisfied."

"$\Longrightarrow$" part. Suppose for a given K, there exists a set $\mid A' \mid\leq K$ such that A' contains at least one arc from every directed cycle in G. It follows that $G'(V, A \setminus A')$ has no cycle. If an order $p : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots, n\}$ is the result of a topological sort on G', then $f(i) \equiv n - p(i) + 1$ is the one-to-one function such that

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} cost_{ij} d_{f(i)f(j)} \leq K.$$

It is because the arcs in A' become forward arcs given order f (or backward arcs given order p), and the number is $\leq K$.

“$\Longleftarrow$” part. Suppose there exists a one-to-one function f such that

$$\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} cost_{ij} d_{f(i)f(j)} \leq K.$$

After removing the arcs (i, j) such that $f(i) < f(j)$ from G, there will be no cycles. Since the number of such arcs is $\leq K$, they form a subset $A' \subseteq A$ with $|A'| \leq K$ such that A' contains at least one arc from every directed cycle in G. □

References

[1] R. Allen and K. Kennedy. Automatic transformation of Fortran program to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[2] R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th Annual ACM Symp. on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[3] R. Anderson, A. Munshi, and B. Simons. A scheduling problem arising from loop parallelization on MIMD machines. In *3rd Aegean Workshop on Computing, AWOC 88,Corfu,Greece*, pages 124–133, June/July 1988.

[4] D.-K. Chen. MaxPar: An execution driven simulator for studying parallel systems. Master’s thesis, University of Illinois at Urbana-Champaign, October 1989. Also available as CSRD tech report No. 917.

[5] D.-K. Chen and P.-C. Yew. An empirical study of DOACROSS loops. In *Supercomputing ’91*, pages 620–632. IEEE Computer Society Press, November 1991. Also available as CSRD tech report No. 1140.

[6] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for DOACROSS loops. In *1994 Int’l. Parallel Processing Symposium*, pages 477–481, April 1994. Also available as CSRD tech report No. 1315.

[7] R. Cytron. *Compile-time Scheduling and Optimization for Asychronous Machines*. PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[8] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Int’l. Conf. on Parallel Processing*, pages 836–845, August 1986.

[9] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[10] V. Krothapalli and P. Sadayappan. Removal of redundant dependences in doacross loops with constant dependences. *IEEE Trans. on Parallel and Distributed Systems*, 2(3):281–290, July 1992.

[11] D. J. Kuck. *The Structure of Computers and Computations*, volume I, chapter 2, page 139. John Wiley and Sons, New York, 1978.

[12] Z. Li and W. Abu-Sufah. On reducing data synchronization in multiprocessed loops. *IEEE Trans. on Computers*, C-36(1):105–109, January 1987.

[13] S. Midkiff and D. Padua. A comparison of four synchronization optimization techniques. In *Int'l. Conf. on Parallel Processing*, volume II, pages 9–16, Aug. 1991.

[14] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of ACM*, pages 1184–1201, Dec. 1986.

[15] D. A. Padua. *Multiprocessors: Discussion of Some Theoretical and Practical Problems*. PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1979.

[16] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, c-29(9):763–776, September 1980.

[17] D. K. Poulsen and P.-C. Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. In *Supercomputing '91*, pages 860–869, November 1993.

[18] E. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall Inc., 1977.

[19] B. Simons and A. Munshi. Scheduling loops on processors: Algorithms and complexity. *SIAM J. of Computing*, 19(4):728–741, August 1990.

[20] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. Matrix eigensystem routines - eispack guide. Heidelberg, 1976.

[21] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comupting*, 1(2), 1972.

[22] M. J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1982.