[25] Joe Warren. A hierarchical basis for reordering transformations. *Eleventh ACM Principles of Programming Languages Symposium*, pages 272–282, January 1984. Salt Lake City, UT.

[26] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop parallelism. *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, August 1990. Also published in *Monograph in Parallel and Distributed Computing*, Pages 243–259, Pitman, 1991.

[27] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 26–28 1991.

[28] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. Revised version of the author's Ph.D. dissertation, Published as Technical Report UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, 1982.

[11] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[12] Guang R. Gao, Russell Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. ACAPS Technical Memo 41, School of Computer Science, McGill University, March 1992.

[13] Allen Goldberg and Robert Paige. Stream processing. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 53–62, Austin, TX, August 1984.

[14] Frank Harary. *Graph Theory.* Addison-Wesley Publishing Co., 1969.

[15] D. J. Kuck, Kuhn R., D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[16] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, Ft Worth, TX, 1976.

[17] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[18] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[19] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, MA, 1989. In the series, Research Monographs in Parallel and Distributed Computing. This monograph is a revised version of the Author's Ph.D. dissertation published as Technical Report CSL-TR-87-328, Stanford University, April 1987.

[20] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. Technical report, IBM Research, October 1990. Submitted to IBM Journal of Research and Development.

[21] Vivek Sarkar. The PTRAN parallel programming system. In B. Szymanski, editor, *Parallel Functional Programming Languages and Environments.* McGraw-Hill Series in Supercomputing and Parallel Processing, 1990.

[22] Vivek Sarkar and David Cann. POSC—a partitioning and optimizing SISAL compiler. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 148–163, June 1990. Amsterdam, the Netherlands.

[23] Vivek Sarkar and Guang R. Gao. Optimization of array accesses by collective loop transformations. *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 194–205, June 1991.

[24] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

by Gao, Olsen, Sarkar, and Thekkath, collective loop optimization was considered both at the individual loop-nest level (e.g., loop reversal and loop interchange) and at a global level, where loops are partitioned into clusters for fusion [12].

8 Acknowledgements

We gratefully acknowledge the significant contributions made by Dr. Vivek Sarkar at IBM Palo Alto Scientific Center and Radhika Thekkath at University of Washington, since the influence of their research is reflected throughout this document. Likewise, we gratefully acknowledge the financial support provided by the National Science and Engineering Research Council (NSERC) and Bell Northern Research (BNR).

References

[1] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, 1983.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[3] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. Also published in *Monographs in Parallel and Distributed Computing*, Pages 192–219, Pitman, 1991.

[5] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.

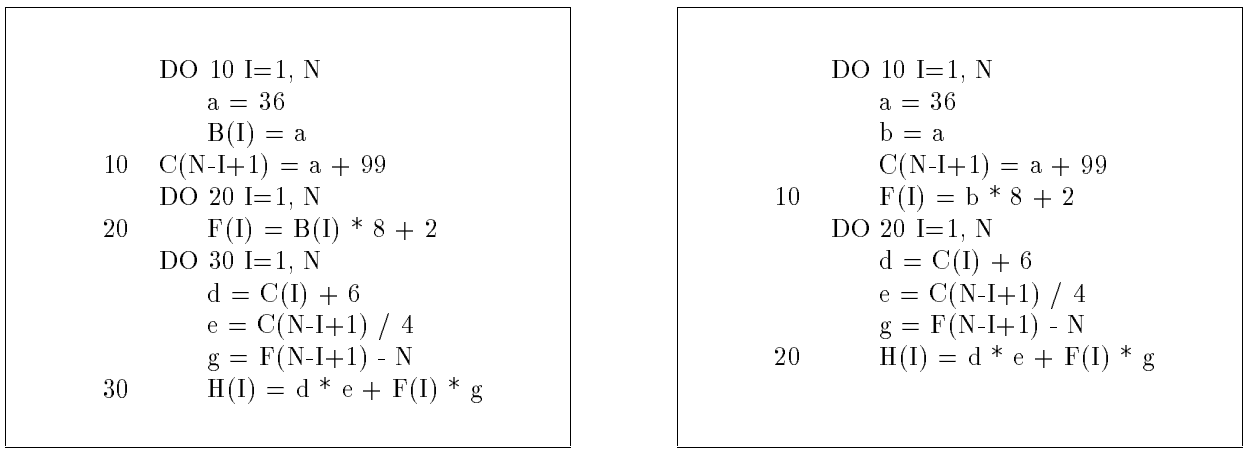
[6] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.

[7] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, January 1981.

[8] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science*. Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[9] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerishe Mathematik*, 1:269–271, 1959.

[10] G. R. Gao. A pipelined code mapping scheme for static dataflow computers. Technical Report TR-371, Laboratory for Computer Science, MIT, 1986.



(a) Non-Optimal Solution **(b) Optimal Solution**

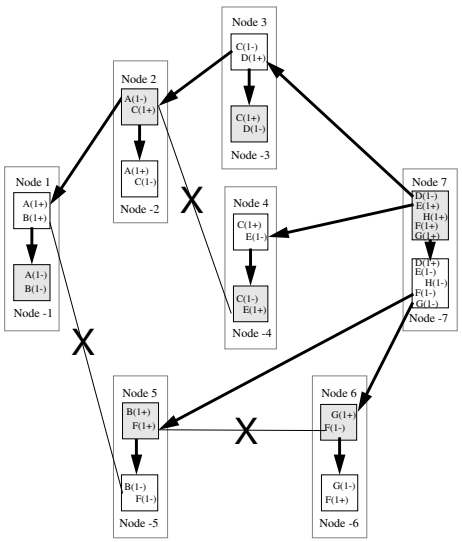
Figure 23: Non-Optimal and Optimal Partitioning Solutions

Loop distribution and loop fusion are well known loop transformations to handle multiple loop nests [28]. Loop distribution is typically performed maximally (Π-block partitioning), and loop fusion is used as follow-up to selectively merge the distributed loops back together. In general, loop fusion reduces loop overhead in uniprocessor codes, increasing the size of loop bodies. This affects instruction-scheduling effectiveness and reduces memory requirements, in turn, reducing memory-register data traffic. This last benefit is especially important for vector machines for which vector loads and stores are costly. In spite of these benefits, the ability to effectively perform loop fusion has remained a difficult problem [1].

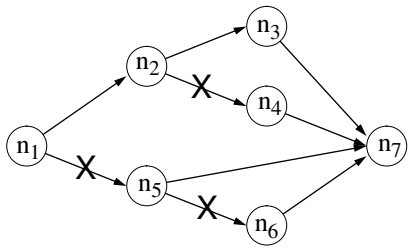
Warren presented an algorithm, based upon loop fusion, which merges a collection of loops, allowing the replacement of temporary arrays by scalars [25]. His algorithm was based upon a hierarchical dependence graph upon which loop fusion can be performed. The method however does not indicate how to perform fusion to minimize the cost of array accesses using scalar replacement. Goldberg and Paige described loop fusion as an optimization technique which attempts to find an "optimal schedule" by minimizing the number of fusable loop clusters [13]. Their work shows the minimization problem to be NP-Hard, and it offers some heuristic solutions.

Wolfe used the term "array contraction" for a different optimization than the one described here. His use of the term was to describe the function of reducing the size of a compiler-generated temporary array created by scalar expansion [28]. Allen also investigated the problem of minimizing temporary array storage in a single loop nest, using a technique called *sectioning*; however, the technique was not extended for optimization beyond a single loop nest [1]. Callahan et al., have studied register allocation of subscripted variables (arrays), but as with the other related research, their work too focused on single loop nests, as opposed to collections of loops [6].

The opportunity for optimizing array operations across a collection of loops for *dataflow software pipelining* was recognized in Gao [10, 11], and loop parallelization and loop chunking across collections of loops has been described by Sarkar [19, 22, 20]. In our work and in recent work

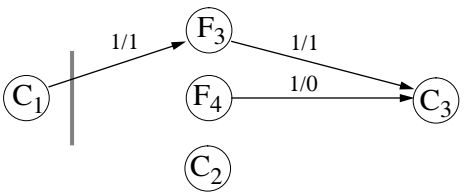


(a) Interference Graph



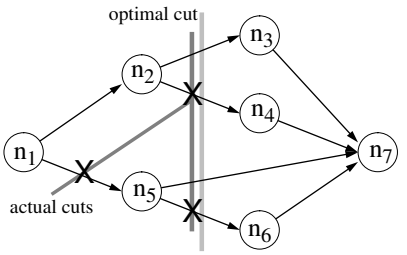
	Feasibility Sets						
Step	F_1	F_2	F_3	F_4	F_5	F_6	F_7
1	{1}	{1}	{1}	{2}	{2}	{3}	{3}
2	{1}	{1}	{1–3}	{2, 3}	{2}	{3}	{3}

(b) Loop Dependence Graph



$$F_3 = \{C_1, C_2, C_3\} \quad F_4 = \{C_2, C_3\}$$
$$C_1 = \{n_{1,n_2}\} \quad C_2 = \{n_5\} \quad C_3 = \{n_6, n_7\}$$

(c) Cluster Graph



(d) Optimal and Non-Optimal Partitions

Figure 22: A Case of Non-Optimal Partitioning

```
      DO 10 I=1, N
          A(I) = 36
10        B(I) = A(I)
      DO 20 I=1, N
20        C(I) = A(N-I+1) + 99
      DO 30 I=1, N
30        D(I) = C(N-I+1) + 6
      DO 40 I=1, N
40        E(N-I+1) = C(I) / 4
      DO 50 I=1, N
50        F(I) = B(I) * 8 + 2
      DO 60 I=1, N
60        G(I) = F(N-I+1) - N
      DO 70 I=1, N
70        H(I) = D(N-I+1) * E(I) + F(I) * G(I)
```

Figure 21: Another Collection of Non-Compatible FORTRAN Loops

added to C_3, since this is the only cluster to which F_4 is connected. Feasibility node F_3, however, could go in either C_1 or C_3. The decision in this case is determined by application of the flow-augmenting path algorithm, which results in a cut between C_1 and F_3, as shown in Figure 20(c). With this cut, F_3 can be collapsed and n_3 added to C_3. The result, therefore, is the partitioning: $C_1 = \{n_1, n_2\}$, $C_2 = \{n_5\}$, and $C_3 = \{n_3, n_4, n_6, n_7\}$, as shown in Figure 22(d).

Observe that the partitioning produced in this instance is non-optimal. As shown in Figure 23(a), three loops were needed and four of the original seven arrays were eliminated; whereas, in the optimal solution, shown in Figure 23(b), only two loops are actually required, and five of the original arrays could actually be eliminated.[13] The reason for the non-optimality of the collective loop fusion algorithm is explained by the fact that the algorithm implicitly assumes that non-compatibility of a non-compatible X-edge is determined entirely by the relationship between the end nodes of the particular edge and is unaffected by other cuts made during the partitioning process, and therefore, fixed prior to partitioning; whereas, in fact, this is not actually the case, as explained in Section 5.

7 Other Related Techniques

Most previous work in optimizing the performance of loops has focused on individual loops rather than on collections of loops [15, 2, 28]. Only recently has a combination of loop optimization techniques been applied together and only in the case of a single loop nest, specifically, unimodular transformations [4, 26].

[13] Incidentally, Algorithm III, described in Section 5 (page 32), is able to find the optimal partition in this case.

all real sources. Similarly, a *pseudo sink* is created having infinite-capacity arcs from all real sinks to the pseudo sink.

6.2.2 Algorithm Complexity

The best way to determine the complexity of Algorithm VI is to examine the complexity of each phase of the algorithm separately. First, let us assume that the input LDG has n nodes and a arcs. Phase 1 is composed of three steps: The first is the forward pass of the nodes which also examines incoming arcs. Since each node and each arc are visited once, $\mathcal{O}(n+a)$ steps are required. The second step is the reverse pass of the graph which visits each node and the arcs leading out of these nodes. In the worst case, all nodes are traversed, and all arcs are examined, requiring $\mathcal{O}(na)$ steps. In the third step the CG is built, requiring a visit to all nodes (to build the node set) and a visit to all arcs (to build the arc set), requiring $\mathcal{O}(n+a)$ steps. The total time is $\mathcal{O}(n+a) + \mathcal{O}(na) + \mathcal{O}(n+a) = \mathcal{O}(na)$. To find the complexity of Phase 2, we assume that in the worst case each application of the flow-augmenting path algorithm isolates only a single node, requiring $n-1$ applications. If the algorithm takes $\mathcal{O}(na \log n)$ [24], the total time is $\mathcal{O}(n^2 a \log n)$. Therefore, the total time for the entire Algorithm VI is $\mathcal{O}(na) + \mathcal{O}(n^2 a \log n) = \mathcal{O}(n^2 a \log n)$.

6.2.3 A Case of Non-Optimal Partitioning

As explained at the beginning of Section 6.2, determining an optimal partitioning is likely to be yet another NP-Complete problem, and although the heuristic algorithm we have just described produces an optimal partitioning in many instances, there are instances which are equally likely to occur for which the algorithm produces non-optimal results. In this section we give one such example. The code for this example is shown in Figure 21. The example loop collection in this case is comprises seven loops and uses eight arrays. The LDG for this code is shown in Figure 22(a). Observe that the collection has three non-compatible arcs, $(n_1, n_5), (n_2, n_4)$, and (n_5, n_6), identified by earlier loop analysis to be non-compatible (as described in Section 2 or in the first part of Section 6.1).

The forward and reverse traversals of the LDG, required in Phase 1 of the algorithm, produces the feasibility sets shown in Figure 22(b). For this particular collection of loops, the algorithm identifies an overall requirement for three clusters. However, this overstates the requirement since only two clusters are actually required for optimal partitioning, as we will show later.

The CG which is formed from the feasibility sets (identified in the table) and from the induced set of arcs from the LDG is shown in Figure 22(c). In this particular CG there are two feasibility nodes, F_3 and F_4, and three cluster nodes, C_1–C_3, created from the fusibility sets, and there are three induced edges taken from the LCG. The arc capacities required to make the graph conformable for network flow analysis are also shown in Figure 22(c), in addition to the cutset resulting after application of the flow-augmenting path algorithm. In this instance, only one iteration of the flow-augmenting path algorithm is needed to partition the graph, and network saturation is accomplished with only one path augmentation. Feasibility node F_4 is immediately collapsed so that n_4 can be

Input: The reduced graph $G' = [V', A']$.

Output: A set of compatible clusters.

Procedure:

FORALL $n' \in N'$
;; collapse and label arcs in G'
  IF multiple compatible arcs (m, n) are incident to n
    delete all but one of these arcs
    flow capacity $c_{mn} = 1 +$ number of (m, n) removed
WHILE $|N'| > 1$ *;;* G' *is connected*
;; remove C_i *from* G'
  execute the *flow-augmenting path* algorithm to determine a partition
  remove the `partition` from G'
  IF the `partition` is $\{C_i, F_n\}$
    add n to C_i
  ELSE
    add n to C_j where $j \in F_n$
  add C_i to the `solution`
  FORALL $F_n \in G'$
    remove C_i
RETURN the `solution`

In Figures 20(d)–(f) we show the application of Phase 2 of Algorithm VI to the CG in Figure 20(c). The capacity and flow following application of the flow-augmenting path algorithm is shown for each arc in Figure 20(d); for each arc, capacity is shown to the left of the slash and flow is shown to the right. Observe that the two arcs between F_2 and C_2 were replaced by a single arc having a capacity of 2. A similar replacement was made between nodes F_5 and C_3. The maximum flow initially occurred between C_1 and F_2, allowing n_2 to be added to C_2 (see Figure 20(d)). The arc between C_2 and F_5 creates another arc between C_3 and F_5, and this is replaced by a single arc of capacity 2. In Figure 20(e) we show the application of the flow-augmenting path algorithm to the remaining graph. This final step places n_5 into cluster C_3, terminating the algorithm.

Certain types of LDGs require special treatment in order to be handled by Algorithm VI. For example, the LDG might contain a set of non-fusible nodes, such as n_1 and n_4 in Figure 20, which each must be put into separate clusters. These nodes are used during the assignment of cluster numbers, but they are not essential to the application of the flow-augmenting path algorithm. Therefore nodes of this type are deleted from all cluster nodes in the CG, as are the arcs to and from these nodes. This isolates these nodes so that the flow-augmenting path algorithm can be correctly applied. For situations in which the LDG is disconnected, Algorithm VI is applied to each component, and once all components are partitioned, the clusters having the same cluster number are merged. Another situation which requires special treatment is one in which there are multiple sources or multiple sinks. In these cases, a *pseudo source* is created having infinite-capacity arcs to

IF $x_m < \texttt{MIN}(x_n)$
FORALL j s.t. $x_m < j \leq x_n$
add j to F_m
;; construct G', having N' and A' defined as follows:
FORALL $F_n \in F$
IF $i \in F_n^1$
$C_i \in N'$, where $C_i = \{n \mid i \in F_n\}$
ELSIF $F_n^{>1}$
add F_n to N'
FORALL $n' \in N'$
IF $\{m, n\} \not\subseteq n'$ **AND** (m, n) is contractable
add (m, n) to A'

The CG initially consists of a single component which is iteratively reduced as compatible clusters are identified and removed. Feasibility nodes in the graph are also collapsed into adjacent cluster nodes, as part of the cluster identification process. To collapse a feasibility node, the node itself is removed and the node in the LDG which it represents, i.e., the node indicated by the F-subscript, is added to the adjoining cluster to which it is being assigned. To ensure cluster assignment is accomplished in a cost effective manner, Algorithm VI uses the flow-augmenting path algorithm to determine to which adjacent cluster node the LDG node referenced by the feasibility node should be assigned. The collective-loop-fusion problem is transformed into a network flow problem by assigning a capacity of 1 to each arc in the CG. Multiple arcs between adjacent nodes in the CG are also collapsed where possible and arc capacity adjusted. The flow-augmenting path algorithm returns a minimum partition containing the fewest number of arcs. The flow-augmenting path algorithm is repeatedly applied on the primary component until the CG is reduced to only a single node.

When the partition isolates a feasibility node, as opposed to a cluster node, the feasibility node becomes a new component in the the reduced CG which continues to be considered each time a new cluster node is removed, possibly removing a cluster reference from the isolated feasibility node. At the stage the feasibility node contains only one remaining cluster reference, it is eliminated, just as other feasibility nodes within the primary component, and the corresponding node referenced in the LDG is assigned to the remaining referenced cluster, just as in the preceding case.

Each application of the flow-augmenting path algorithm partitions the graph into two components, each smaller than the original. Sometimes the CG can be partitioned simultaneously into more than two components by cutting the graph at every arc for which the flow capacity has been reached, not just the minimum cutset. Two cases in which the graph cannot be cut in this way are: *1)* when a cluster node is disconnected from the other nodes, and *2)* when removal of an arc would not disconnect the graph. The following formally describes Phase 2 of the algorithm:

Algorithm VI (Phase 2) *Partition a Cluster Graph (CG) into compatible clusters*

The complementary set of feasibility sets are those containing nodes that could belong to more than one cluster; these sets are indicated by $F_n^{>1} = F - F_n^1$, where $F = \{F_n \mid 1 \leq n \leq |N|\}$. All of the consolidated nodes $n' \in N'$ in G' are created from either one or the other of these two types of sets. The first type of node in N' actually represents a cluster of nodes from G, indicated by $C_i = \{n \mid n \in F_n^1\}$. For example, in Figure 20(c), cluster node $C_2 = \{n_3, n_4\}$ was created from feasibility sets F_3 and F_4, cf., Figure 20(b). The second type of node in G' is a node F_n, formed from the corresponding feasibility set $F_n^{>1}$ for n. In Figure 20(c), for example, feasibility node F_2 corresponds to feasibility set F_2 in Figure 20(b), and likewise node F_5 corresponds to set F_5. The set of arcs in G' is $A' = \{(m,n) \in N' \times N' \text{ s.t, } \{m,n\} \not\subseteq C_i \text{ and } (m,n) \text{ is contractable}\}$. The arcs not included in N' therefore are *1)* those that are eliminated because they would be internal to one of the newly formed cluster nodes and *2)* those that are eliminated because their edge-capacity is zero, such as non-fusible arcs (marked by an asterisk).[12]

Overall, the construction of the cluster graph achieves two purposes: *1)* it consolidates nodes in the LDG that must belong to the same cluster, and *2)* it eliminates superfluous arcs that are not required in making future partitioning decisions. The CG is also the input for the Phase 2 transformation, which we describe next. Before we describe Phase 2, however, we formally state Phase 1:

Algorithm VI (Phase 1) *Create a Cluster Graph from a Loop Dependence Graph (LDG).*

Input: An acylic graph LDG $G = [N, A]$.

Output: An acyclic cluster graph $G' = [N', A']$.

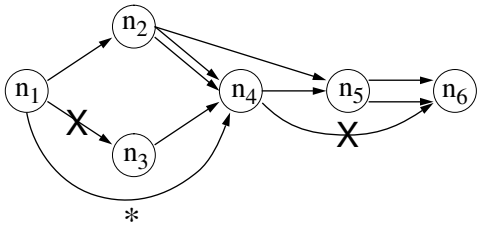
Procedure:

;; traverse the nodes of G in topological order
FORALL $n \in N$
IF n has no incoming arcs
$F_n = \{1\}$
ELSE
FORALL input arcs (m,n) incident to n

$$F_n = \texttt{MAX}\left(\left\{ \begin{array}{ll} \{i+1\},\ C_i \in F_m, & \text{if } (m,n) \text{ is non-compatible} \\ F_m, & \text{otherwise} \end{array} \right.\right)$$

;; traverse the nodes of the graph in reverse topological order
FORALL $m \in N$
IF $\exists (m,n)$ s.t. (m,n) is non-compatible
$x_m = \texttt{MAX}(i \in F_m)$
FORALL output arcs (m,n) incident to m
$x_n = \texttt{MAX}(j \in F_n)$

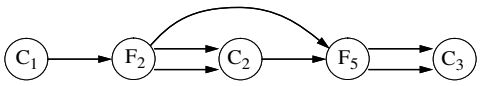
[12] Recall that non-fusible arcs are also non-contractable.



(a) Loop Dependency Graph (LDG)

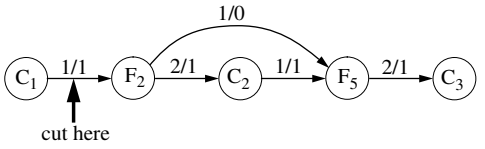
	Feasibility Sets					
Step	F_1	F_2	F_3	F_4	F_5	F_6
1	$\{1\}$	$\{1\}$	$\{2\}$	$\{2\}$	$\{2\}$	$\{3\}$
2	$\{1\}$	$\{1,2\}$	$\{2\}$	$\{2\}$	$\{2,3\}$	$\{3\}$

(b) Feasibility Sets

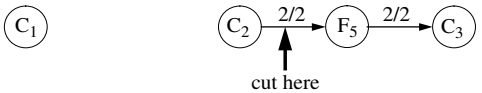


$F_2 = \{C_1, C_2\}$ $F_5 = \{C_2, C_3\}$
$C_1 = \{n_1\}$ $C_2 = \{n_3, n_4\}$ $C_3 = \{n_6\}$

(c) Cluster Graph



(d) Cluster Graph Reduction



(e) Follow-on Reduction

$C_1 = \{n_1\}$ $C_2 = \{n_2, n_3, n_4\}$ $C_3 = \{n_5, n_6\}$

(f) Final Solution

Figure 20: An Example of Cluster Partitioning

sequence to determine the earliest cluster to which each node could belong and then scanning the nodes in a reverse topological sequence to determine the remaining clusters to which each could belong. The second phase of the algorithm is also a two-step process: First, nodes which could only belong to a single cluster are removed from the sets of feasible clusters and assigned immediately to an actual cluster. This leaves the nodes which could belong to more than one cluster. For these nodes, a least-cost determination is made as to which actual cluster each node should be assigned. This later determination is made in a iterative manner using the flow-augmenting path algorithm, which is based upon the max-flow min-cut theorem (described in Section 5).

6.2.1 Terminology and Methodology

Before formally presenting the algorithm for collective fusion, we need to explain some additional terminology and give a general description of the overall methodology.[11] The input to the algorithm is an LDG, say $G = [N, A]$, in which all nodes $n \in N$ have been assigned unique labels according to their topological order. As indicated above, the first step is to determine the *feasible clusters* to which each node might belong. This information forms a set $F_n = \{c \mid 1 \leq c \leq k\}$ for each node, where n is the topological number of the node, c is the number of a cluster to which n could belong, and k is the minimum number of clusters that are necessary to partition the non-compatible loop collection represented by G. To determine the members of these *feasibility sets*, the algorithm first visits each node in topological sequence and determines the lowest-numbered cluster to which each node can belong. For source nodes in G, $F_{src} = \{1\}$, indicating the earliest cluster to which each can be assigned is cluster C_1. If the input arc of a node is a non-compatible arc (marked with an X), the earliest cluster is one more than the cluster number at the origin of that arc. Likewise, if a node n is a terminus to multiple arcs, the highest cluster number of any origin incident an input arc of n determines F_n. If the maximum cluster number assigned to any feasibility set is k, then k is the minimum number of clusters required to partition the graph. In the table in Figure 20(b) we show for Step 1 the feasibility sets for each node following forward traversal of the LDG in Figure 20(a). In this case, G requires three clusters, C_1, C_2, and C_3.

As a follow-on step, the nodes of G are traversed in reverse topological sequence to determine the last and any possible intervening clusters to which each node could belong. These cluster numbers too are added to the feasibility sets of the respective nodes. As a result of this step, the feasibility set for any particular node can contain several possible clusters. Multiple node feasibility sets can be seen, for example, in Step 2 of the table in Figure 20(b); they are sets F_2 and F_5.

Feasibility sets are used to create a reduced graph $G' = [N', A']$, called a *Cluster Graph* (CG); G' is constructed from G by consolidating nodes and arcs in G. In Figure 20(c) we show the CG for the example in Figure 20(a). The nodes in a CG represent two distinct situations: *1)* nodes that have already been been assigned to clusters and *2)* nodes awaiting to be assigned. Each of these categories is represented in the CG by a different type of node. The first type is initially formed from singleton feasibility sets. These sets contain only a single cluster indicating there was no alternative cluster to which the node could be assigned; we indicate sets of this type by $F_n^1 = \{F_n \mid |F_n| = 1\}$.

[11] Our terminology and methodology are somewhat different from that described by Gao, Olsen, Sarkar, and Thekkath [12].

```
      DO 10 I=1, N
10        A(I) = E(I)
      DO 20 I=1, N
          B(I) = A(I) * 2 + 3
20        C(I) = B(I) + 99
      DO 30 I=1, N
30        D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40        E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
          F(I) = B(I) * 4 + 2
50        G(I) = E(I) * 8 - F(I)
      DO 60 I=1, N
60        H(I) = F(I) + G(I) * E(N-I+1)
```

Figure 19: Sample Collection of Non-Compatible FORTRAN Loops

nodes $n_1, \ldots, n_k$ are numbered according to the topological execution ordering of the loop nests in the original loop collection. Also, since the loop nests $n_1, \ldots, n_k$ are assumed to be identically control dependent, there can be no control branching among these loops.

As a means of illustration we will use the collection of FORTRAN loops shown in Figure 19. The LDG for this collection is shown in Figure 20(a). Non-compatible arcs are marked in figure with an X. The non-compatible arc from n_1 to n_3 is due to array `A` being produced in one order and consumed in the reverse order, and similarly, the non-compatible arc from n_4 to n_6 occurs due to array `E` being produced and consumed in reverse orders. The non-contractable arc is marked with an asterisk. A non-contractable arc is one representing an array which cannot be contracted even when loop fusion is possible. The non-contractable arc from n_1 to n_4 is due to an anti dependence on array `E`. All other arcs in the LDG are both compatible and contractable. Collective loop analysis, as described earlier in this section or as described in Section 2, is used to determine arc compatibility, and data dependence analysis, as described by Wolfe, is used to reveal array contractability [28].

Collective loop fusion works on any acyclic graph having an unrestricted number of non-compatible and/or noncontractable arcs. Although the general collective loop fusion problem is probably NP-complete, the algorithm we describe is optimal for bi-partitioning—a case for which the *flow-augmenting path* algorithm yields a cutset of minimum size [12]. The objective of collective loop fusion is to find the minimum number of clusters in which all non-compatible arcs are separated across clusters. According to Gao, Olsen, Sarkar, and Thekkath, this will maximize the number of contractable edges within each cluster [12]. The overall strategy consists of two phases: *1)* determining a set of clusters to which each node could feasibly belong, and *2)* selecting nodes from these sets and assigning them to actual clusters. This later phase is then repeated until all nodes have been assigned.

The first phase of the algorithm is accomplished by scanning nodes first in a forward topological

let G' be a balanced graph and $w_p()$ and $w'_p()$ be functions which return the heaviest paths for G and G', for any node i, $w_p(i)$=$w'_p(i)$ must hold. Without loss of generality, we can assume that G has a unique source node s. The first statement in the buffer-allocation algorithm (below) finds the heaviest path weight from the source to all other nodes, using a technique such as Dijkstra's shortest-path algorithm [9].

Algorithm V *determine requirements for external buffers*

input: A weighted LCG, $G = (N, A, W)$ having a unique source node s.

output: A balanced graph $G' = (N, A, W')$.

procedure:

compute the heaviest-path $w_p()$ for G
FOR each arc $a = (i, j) \in A$
    insert a buffer of size $w_p(j) - w_p(i) - w_a$ along arc a
    add this size to w_a to obtain $w'_a = w_p(j) - w_p(i)$, the new weight in W'
RETURN G'

The overall complexity of the shortest-path algorithm is $\mathcal{O}(|N||A|)$, but since node degrees are usually small, $|A| \simeq |N|$. Therefore, for practical purposes the algorithm has a complexity of $\mathcal{O}(|N|^2)$.

6.2 Collective Loop Fusion

Collective loop fusion is a loop partitioning technique which attempts to find a set of loop clusters which minimizes the number of inter-cluster arcs. It is an alternative technique to the partitioning technique described in Section 5. Although the technique is oriented toward uniprocessor applications, the same performance benefits accrue for multiprocessor applications as well, especially if limited resources will not permit complete distribution of the original loops among processors. The objective of collective loop fusion implicitly assumes a cost savings from each compatible-contractable arc. For example, assume this savings is 1 unit, denoting the cost of an array-element load from memory. The storing cost is ignored because *1)* whether it can be eliminated depends upon whether the element is used later in the program and *2)* data storage is buffered in most modern uniprocessor architectures. It is also possible for a store or a load to be contained within a conditional, in which case the arc cost would be less than 1, based upon the branching probabilities of the respective conditionals [21].

The program representation used in the following algorithm is a *Loop Dependence Graph* (LDG). An LDG is essentially an LCG, with the following differences: The arcs in an LDG correspond to *flow*, *anti*, or *output* data dependences [28], and only flow arcs can be compatible. In an LDG,

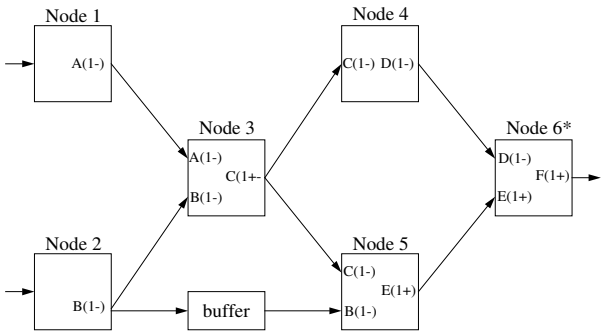


Figure 18: Buffer Allocation for an LCG

6.1.2 Buffer Allocation

Buffer optimization is performed only on compatible loops. We illustrate buffering with our original example from Section 2, Figure 1, page 7. Recall array `B` is generated by loop 2 and is consumed along two different paths. On the upper path, `B` is processed first by loop 3, and the result, array `C`, is sent on to Loop 5. On the lower path, `B` is routed directly to loop 5. Loop 2 could run to completion before loops 3 and 5 begin, but this would cause the entire array to be stored. On the other hand, the producer (loop 2) and the consumers (loops 3 and 5) are able to execute concurrently [11]. As the elements of `B` are generated by loop 2, they can be immediately consumed by the follow-on loops. Again, this can only occur if the array communication arcs are compatible. If the LCG is cyclic, buffers are used only on the forward arcs and not on feedback arcs.

Since the two paths in Figure 1(b) have different execution times, temporary storage is needed along the lower communication path to hold elements of `B` until they are required by loop 5. In Figure 18 we show the LCG after transformation. In this case, a buffer of size w keeps the three loops operating in a maximally pipelined manner, where w is a function of the execution time of upper path.

To solve the buffer-allocation problem, it is necessary to assign weights to arcs in the LCG. Let $G = (N, A, W)$ be a weighted LCG with each arc $a = (i, j) \in A$ being weighted by a nonnegative integer $w_a \in W$. When a value produced by node i arrives at an input port of j, it is retained for some number of instruction executions before reaching the output port—this number is w_a.

The objective of buffer allocation is to determine requirements for buffers such that the weight of any two distinct paths between any two nodes are equal. An LCG which meets this condition is *balanced*. The algorithm to accomplish the balancing does two things: 1) it locates arcs where buffers are needed, and 2) it determines the size (or weight) of each required buffer. In solving the problem we observe that the heaviest path from node i to j must not change, and that buffers must be inserted only up to the point that all path weights between i and j are the same. As part of the procedure, the weight of the heaviest path from s to i is determined for each $i \in N$. If we

Algorithm IV *Two-color a Loop Communication Graph (LCG) to minimize the number of non-compatible output ports*

input: An LCG.

output: A loop configuration vector, $(s_1, \ldots, s_k)$, where s_i = "+" if node n_i^+ in the corresponding interference graph (IG) was colored with color c_1 of two colors, and s_i = "−" if node n_i^- in G_I was colored with c_1.

procedure:

construct an IG from the LCG *;; see Algorithm I, page 9*
FOR each edge in the IG
  IF edge e_{ij} is an $\{n_a^+, n_a^-\}$ edge *;; if e_{ij} connects nodes within a block*[10]
    weight w_{ij} is $+\infty$
  ELSE
    w_{ij} is number of static references within the node at the input port
`bestcost` := $+\infty$
FOR each node i in the IG
  build a maximum-weight spanning tree T_i, rooted at node i
  color T_i with two colors, starting at a precolored node
  `curcost` := sum of the weights of all edges connecting the same-color nodes
  IF `curcost` < `bestcost`
    `bestcost` is `curcost`
    store the coloring in `colormap`.
RETURN `colormap` to define the output loop configuration vector

With the Sarkar-Gao heuristic, edge weights are used to guide the construction of the tree, that is, the edge with the largest weight is always the next edge to be included in the tree. To avoid biases due to the choice of root, the spanning-tree algorithm is repeated for all nodes, giving the algorithm a worst-case execution time of $\mathcal{O}(|N_I| \times (|N_I| + |E_I|))$. Compared with Algorithm II, the breadth-first spanning-tree algorithm described in Section 2 (page 11), the Sarkar-Gao technique, using Algorithm IV, is quite costly; recall Algorithm II executes in $\mathcal{O}(N)$ time. The difference in cost is justified in the multiprocessor case, however, since the follow-on transformations too are quite different. In the uniprocessor case, the objective is loop fusion and array contraction; therefore, the least-cost arcs to remove in the case of non-compatibility are always those which form the branches of the forks in the LCG. On the other hand, forks have no special significance with respect to the follow-on distribution of loops among processing elements in the multiprocessor case, and so the number of alternatives for arc selection is much greater, and the added complexity is useful. Algorithm IV also handles cyclic interference graphs, and hence it is suitable for cyclic LCGs. However, the feedback edges must initially be compatible; otherwise the original program will deadlock. To avoid deadlock, the IG edges corresponding to LCG feedback edges are each assigned a cost of $+\infty$.

[10] The definition of a block within an IG is given in Section 2.2 (page 7).

6 Related Research

In this section we describe alternative methods for determining loop compatibility and for partitioning non-compatible loops into compatible loop clusters. First we examine the collective loop-analysis and transformation technique proposed by Sarkar and Gao [23]. This technique provides the basis for the technique we described in Section 2. The second technique we describe is the collective loop fusion method for non-compatible loop clusters described by Gao, Olsen, Sarkar, and Thekkath [12]. This later technique addresses the partitioning problem examined in Section 5. Lastly, we provide a brief review of other related research on the general transformability of loops.

6.1 Loop Analysis and Transformation

The Sarkar-Gao method for collective loop analysis is more general than the method described in Section 2 because the target computer architecture can be either a uniprocessor or a multiprocessor, provided the multiprocessor has sufficient processing elements, inter-processor communication channels, and facilities for small communications buffers at the communication ports of each processing element. In the above context, "sufficient" implies the architecture has adequate resources to take advantage of the parallelism exposed by the transformations; therefore, to a compiler the machine is viewed as having unbounded resources. Although unrealistic, this assumption is usually not restrictive since the required parallelism is usually within the capability of modern multiprocessor architectures [11].

For uniprocessors the transformation strategy is to create fusionable loop clusters, as described in Sections 2–5; but the algorithms described in this section differ slightly, and the impact of these differences is examined. As described in Section 6, however, software pipelining and other instruction scheduling techniques are then used to exploit the instruction-level parallelism. For multiprocessors the strategy is to situate producer and consumer loops on separate processors with a physical communications buffer between processors to provide a maximal flow of array elements to each processor, without using main memory for intermediate storage. In this case, loop transformation is performed in two steps: *1)* loop nests are transformed to maximize loop compatibility, thereby permitting the pipelining of array elements between loops, and, *2)* buffer requirements are determined which maintain a maximum flow of array elements between producer and consumer loops. For the remainder of this section we focus primarily on the multiprocessor case.

6.1.1 Loop Compatibility Analysis

Sarkar and Gao were the first to formulate the loop compatibility problem as a two-color graph coloring problem, using an interference graph (IG) constructed from a loop communication graph. They also proved that, if a solution exists to the two-coloring problem, it can be used to obtain an optimal loop configuration, and they described a heuristic to find a two-coloring of the reduced graph obtained by selectively removing IG edges. The particular algorithm they proposed for determining loop compatibility is given below:

deeply nested/concurrent cycle, unless flow is prematurely restricted by a prior odd-length cycle. For just a moment, neglect the possibility of serial cycles, and consider only adjacent cycles. The partitioning algorithm always breaks the most deeply nested odd-length cycle at its fork, the least-cost location. This break has the added effect of severing one of the two segments in an outer-level cycle, and obviously, the same is true of the second segment of the outer cycle, if an odd-length nested cycle is present along that segment. It too is cut prior to the outer-level fork, thus ensuring the outer-level cycle is cut at its least-cost location, at the same time. Whether the outer-level cycle has an X-arc and zero capacity arc at the fork or not, has no effect in this instance because of the existence of higher-priority cuts prior to the fork.

Now let us consider serial cycles. As previously emphasized, the flow-augmenting path algorithm cuts the graph in a greedy fashion when multiple flows simultaneously reach capacity at several locations at once. Such cuts always occur within the odd-length cycle, or odd-length adjacent cycles, closest to the source node of the graph. As a consequence, only nodes on the far side, or sink side, of the partition need be searched for additional serial cycles. At the same time, serial cuts honor the same priorities described in the preceding paragraph with regard to internal cycles. Therefore, they too are accomplished in an least-cost manner. As a further consequence of serial partitioning, however, the flow-augmenting path algorithm must be run again, once for each serial combination of odd-length cycles remaining in the graph. Since doing so is unavoidable, and since each subsequent partitioning is guaranteed to be accomplished in the same least-cost manner, the partition algorithm is likely to always find the best place to partition the graph during each succeeding recursion.

As a final step, let us assume we have an arbitrary spanning graph which for which we have established an optimal partitioning, using Algorithm III. If we were to now connect any primary node within this graph to any other node in the graph, the connection would result in one or more new cycles. However, the creation of these new cycles would in no way affect the underlying priority relationships described above. In other words, the above partitioning priorities would remain invariant: Inner most nested/concurrent cycles would always be cut first, and serial cycles would continue to require multiple partitions. Likewise, the capacity weighting scheme and network saturation pattern of the flow-augmenting path algorithm would remain invariant, ensuring that the priority relationships manifested in the new graph too would be respected by the cluster partitioning algorithm. As a consequence, we have strong evidence to suggest that Algorithm III is, in fact, an optimal algorithm. Nonetheless, the relationships involved in graph partitioning are complex (as previously noted), and additional analysis and testing are required to substantiate our conjecture that Algorithm III is optimal—work which might be undertaken as a follow-on to this research.

Lastly, let us examine the complexity of the algorithm. In effect there are two principle parts: graph coloring and graph partitioning. The first, graph coloring, uses a breadth-first traversal of the interference graph to determine loop compatibility, requiring $\mathcal{O}(n+a)$ steps, where n is the number of nodes in the LCG, or loops in the original cluster, and a is the number of arcs in the LCG, or array transfers between loops in the original cluster. The second part, graph partitioning uses the flow-augmenting path algorithm which requires $\mathcal{O}(na)$ steps. Since a coloring and partitioning occur each time the graph is partitioned, the overall complexity of cluster partitioning is $\mathcal{O}(n(na+a))$.

We are now ready to see how these partitioning priorities are manifest in the flow capacities assigned to the arcs of the SG. In effect, the priority of cuts in the SG is a function of two factors: *1)* the flow capacity on the respective edges, and *2)* the topological order in which odd-length cycles are discovered by the flow-augmenting path algorithm. To see how these factors affect the partitioning, let us reexamine the way odd-length cycles are identified and the way in which flow capacities are assigned.

The way the algorithm identifies odd-length cycles is by their respective forks. The fact that the breadth-first traversal of the IG which is used to create the SG always finds the root is not difficult to show: merely perform a breadth-first traversal of any cycle in the graph, beginning from a join node and traversing the graph in reverse direction from the direction of the arcs.[9] If a cycle exists, the separate paths extending from the join node must come together; otherwise, the structure would not be a cycle, and the only possible place these segments can possibly join is at the fork. By not connecting X-arcs to the SG at the fork and by assigning zero capacity to the adjacent arcs, we ensure all odd-length cycles in the graph are broken. This step alone is enough to guarantee partitioning effectiveness, but it isn't enough to ensure optimality. Note, however, that X-cuts always occur at the farthest end of each segment, and the flow-augmenting path algorithm exploits this fact.

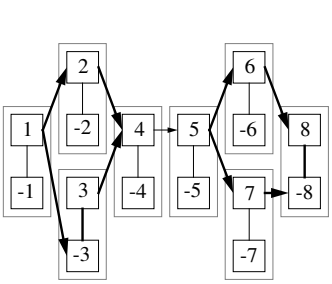
The way optimality is achieved, that is, if it is achieved, is by ensuring that a higher priority cut along a segment of an odd-length cycle is always reached by the flow-augmenting path algorithm prior to reaching the lower-priority X-cut at the farthest end of the cycle. As a consequence of this, lower-priority cuts which are "scheduled" to occur later the the cycle, such as as the X-cuts at the fork, are prevented from occurring, whenever a higher-priority cut exists along the same segment.

Flow capacities also have a lot to do with which arcs are cut: *1)* they ensure arcs at the most deeply nested/concurrent odd-length cycle are broken before any outer-level odd-length cycles, and *2)* they break arcs at the forks if the cycle has not already been previously broken. To verify this fact, one need only observe that the fork in an outer cycle is always farther away from the source node than any fork nodes of a corresponding inner cycle. The two characteristics above are exactly the priorities that are necessary for optimality, as we show next.

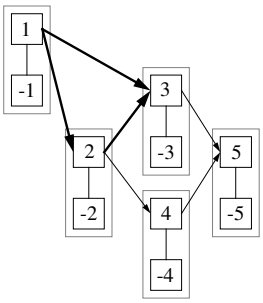
The reason the flow-augmenting path algorithm is able order the priority of cuts stems from the fact that the SG is an acyclic directed graph which imposes a fixed order upon the way the segments of the graph are traversed in search of augmenting paths As a result, if a graph admits multiple minimum-capacity cutsets, the augmenting-path algorithm always saturates arcs closest to the source first. It is this characteristic of the flow-augmenting path algorithm that the partitioning algorithm exploits, and it is a direct consequence of the way flow capacities are assigned. Cuts which have no meaning in the context of the loops themselves are prevented from occurring by assigning them infinite capacities; such as the arcs which connect the primary node to its respective complement node in a block. Likewise cuts which should never be made along a chain because of other higher-priority edges closer to a the fork are also prevented in the same way.

At the same time, the node-capacity labeling methodology ensures no cycle is starved of capacity, unless it is a candidate for cutting. Specifically, it assures sufficient flow capacity reaches the most

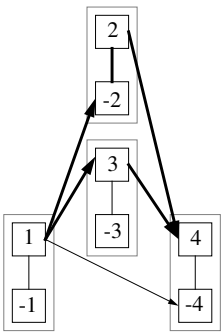
[9] the edge within each block which connects a primary to a compliment node can be viewed as bi-directional.



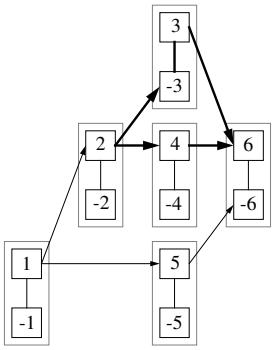
(a) Serial Cycles



(b) Overlapped Cycles



(c) Nested Cycles



(d) Concurrent Cycles

Figure 17: Examples of Ways Cycles Combine

2. If there exists an outer-level odd-length concurrent cycle which has an inner odd-length cycle within one of its segments, the inner cycle must be broken before the outer cycle since the inner cycle must be cut regardless, cf. Figures 17(c) and (d). Moreover, the breaking of an inner cycle always affects the least-cost partitioning of the outer cycle. Consequently, the cutting of inner-most cycles warrant as much priority as the breaking serial cycles.

3. The cuts of least priority are those that are part of overlapped odd-length cycles, the reason being that cutting such cycles affects fewer cycles in general than other combinations of breaks, cf. Figure 17(b). Also, the least-cost break of overlapped cycles is a cut across the arcs of their respective forks, just as it is in the case of a single cycle (as described above). The least-cost cut is never one across a shared arc, as one might suppose. This is because cutting the shared arc still leaves two remaining arcs to be cut. Thus, the total cost is a minimum of three arrays; whereas, a cut across forks of the respective cycles cuts four arcs—but only affects two arrays.

4. Lastly, in the absence of any of the above global criteria, the least expensive location to break an odd-length cycle is at its fork, as explained in the preceding paragraph. It must be emphasized, however, breaking a cycle at the fork is a break of "last resort". Even so, it must be accomplished in order to break the odd-length cycle if no higher priority cut exists.

any other cut affects at least two arrays. Again we emphasize that our ultimate objective is to eliminate as many arrays as possible.

In most instances, the array generated within a fork loop is used by multiple adjacent loops; hence, by partitioning the cycle at the fork, two edges are cut at a cost of one array. On the other hand, a partition anywhere else in the cycle implies the generation of at least two arrays, within two loops, each array being passed to its respective consumer loop; hence, two arcs are cut at a cost of two arrays. Even when the edges of a fork represent the transfer of different arrays, the fork-partition cost can be no higher than any other combination of cuts. Moreover, the preceding least-cost relation holds regardless of how many times a particular array is referenced within a subsequent loop, since the cost of multiple internal array references can always be avoided by making the first use within the loop an assignment to a scalar variable, and the scalar then used to satisfy the remaining references to the array element. Based upon the preceding observations, we conclude that, in the absence of any global criteria, the best location to partition a cycle is across the output edges of a fork.

Now let us take a somewhat broader perspective. First we observe that topologically cycles can be viewed as combining in one of several ways: serially, in a nested manner, in an overlapped manner, or concurrently.[8] An example of each of these basic structural combinations is illustrated in Figure 17. When cycles combine *serially*, the fork node and join node of the first cycle occurs topologically before the fork node and join node of the second, as shown in Figure 17(a). In the figure, two odd-length serial cycles appear, each with bold arcs. When cycles are *overlapped*, as shown in Figure 17(b), the fork of the first is constrained to occur before the fork of the second, and the join of the first is constrained to occur before the join of the second. In this figure, as in the previous one, an odd-length cycle is indicated by bold arcs. When cycles are *nested*, the fork node and join node of the first cycle are constrained by data dependence to occur between the fork and join of the second cycle, as shown in Figure 17(c). Again, the odd-length cycle is highlighted in the figure. Lastly, when cycles occur *concurrently*, there is a nested constraint on the sequence of forks and joins between the two cycles, as shown in Figure 17(d), but the order of internal nodes with respect to an outer cycle is not fixed. Broadly speaking, the cycles in the last three figures all have one thing in common—at least one shared node that is not simultaneously a join in one cycle and a fork in another, and collectively we refer to cycles of this type as *adjacent* cycles.

From examination of Figure 17, a few important relationships become clear:

1. Serial cycles require multiple partitions because there is no way to partition a serial combination of odd-length cycles with a single break. Moreover, since serial odd-length cycles involve multiple breaks, such breaks must be accomplished sequentially, either from left to right or from right to left, to reduce the search cost associated with finding other serial cycles, should any remain. As a further consideration, a side effect of partitioning is usually the breaking of additional cycles which otherwise would cause compatibility constraints on the far side of the partition. These last two points will become clear when we later justify the behavior of the labeling portion of the overall algorithm. Because of these reasons, serial cuts must be given highest priority, if both optimality and efficiency are to be achieved.

[8] These classifications are merely illustrative, since classifying cycles in a graph is like finding images in ink spots.

with several examples illustrating alternative situations which might be encountered. As a final step, we now justify the effectiveness of our loop partitioning method and derive its complexity.

In general, loop partitioning is a difficult problem, and the conjecture is that cluster partitioning too is NP-complete [12]. Nevertheless, the heuristic we described in Algorithm III works well in many instances, as we have shown. Unfortunately, we cannot guarantee this approach produces optimal results in all cases. In the analysis that follows, we therefore present intuitive arguments for why our algorithm frequently succeeds.

We know the flow-augmenting path algorithm for max-flow min-cut produces an optimal partition of a network for a given set of fixed flow capacities. Therefore, if an SG can be effectively transformed into a network with fixed flow capacities assigned in such a way as to accurately reflect the relative cost of loop-cluster partitioning, we are similarly assured the resulting partitioning too is optimal. To establish optimality, we would need to show *1)* our priorities for cutting edges accurately reflect actual partitioning tradeoffs, and *2)* these priorities are respected during application of the flow-augmenting path algorithm. Since we have no evidence to suggest otherwise, we begin under the assumption that the algorithm might, in fact, be optimal. If it is not, we would like to isolate situations in which the algorithm is likely to fail. It is with this motivation that our analysis proceeds.

In the discussion that follows, a *fork node* in the IG is a node with multiple output ports, and likewise a *join node* is a node with multiple input ports. Note that forks in the IG correspond to joins in the SG, and visa versa. We refer to a combination of a fork node, a join node, and the two segments of the graph which connect these nodes as a *cycle*.[7] Cycles in an IG are further classified according to whether or not they are minimal; those consisting of a unique fork and the fewest number of nodes are said to be *fundamental*. Lastly, when we refer to the *segments of a cycle*, it will be to the various chains that connect a fork and a join. Note, as a minimum, every cycle contains at least two such segments. As further terminology, we distinguish undirected edges in the IG from directed arcs in the induced SG.

By construction, we guarantee that the SG has no cycles of odd-length, for we refuse to add arcs to the graph which would produce such cycles. Nonetheless, let us assume for a moment we were to add arcs which would introduce odd-length cycles. Surely we would have to break all of these cycles in order to make the SG two-colorable. At the same time we might need to cut other arcs of the graph to partition it. Partitioning, as we have shown previously, is essential to form fusionable loop clusters. Of course, arcs cannot be cut indiscriminately. Consequently, we need a means of establishing priorities with respect to the order in which arcs of the SG can be cut.

As a way of discovering these priorities, let us assume that an IG consists of just a single isolated odd-length cycle. To break this cycle, we must partition it into two sets of nodes. Moreover, we know by inspection that both segments must be broken, and this represents the least cost. Here each segment consists of, at most, one chain. Obviously, the least-cost cut of each segment, in this case, would be an edge-cut. But, for the case of an isolated cycle, no edge costs less to cut than the edges extending from the fork. This is because usually only a single array is affected; whereas,

[7]This definition is slightly different than common terminology in order to take advantage of both the directed nature of the SG, as well as the characteristics of the underlying IG.

Notice in Figure 16(b), how we depict the situation in which multiple arrays are passed between code blocks, for example, arrays F and G being passed between Blocks 5 and 6. Each array is represented individually since a separate cost is associated with each. In this case, the capacity of Node 5 is 2, and the capacity of each connecting input arc 1, as shown. Next, notice that the arc adjacent to each X-edge has its capacity set to zero, but only for those edges corresponding to the same array. For example, the capacity of arc $(6, 4)$ for array E is zero, corresponding to X-edge $(4, -5)$ for array E, and the capacity of one of the two adjacent arcs $(-4, 2)$ is zero (for array B), but not the other (array C). Consequently, the capacity of Node 2 is 1, not 2. Next, notice the first cut in the SG is made without a single flow-augmenting path ever being made. Also notice that following the initial partitioning, the capacity of all adjacent arcs affected by the initial cut remains unchanged, since the cost of cutting the arc remains valid. In other words, the zero capacity on arc $(-4, 2)$ for array B is not reevaluated for the second iteration of the flow-augmenting path algorithm on subgraph $\{1, 2, 3, 4\}$; whereas the remaining node and arc capacities are (In the figure the revised node capacities in the middle partition are denoted with a hash mark.). The second algorithm application cuts the graph a second time, as shown, resulting in two cuts and three subgraphs overall, cf. Figure 16(c). Although the partitioning results consist of three fusionable partitions, instead of two, the benefit of transformation remains high. In this case, five of the original eight arrays were still able to be eliminated, along with the loop overhead from three loops.

So far we have not discussed cluster partitioning for the case of non-compatible loop clusters caused by constrained loops that are separated an odd number of arcs from one another.[6] In situations of this type, a least-cost cut must be found between the two loops. However, since this type of non-compatibility can often be broken at the same time a cluster is being partitioned to remove the previous type of non-compatibilities, we do not attempt to remove any constrained-loop non-compatibilities until all of the other type of non-compatibilities have been removed (using Algorithm III). If after the other non-compatibilities have be removed, a partition or partitions remain non-compatible, we sum the number of unique arrays referenced for nodes of each color within each remaining non-compatible partition and then, remove from each of these partitions the colored nodes corresponding to the fewest number of references. The effect is to increase the number of arrays required by the number arcs into and out of these nodes.

In Figure 16, if Nodes 3 and -4 were each constrained, Node 3 would be removed from the cluster since the fewest number of arrays in the middle partition would be affected, in this case, two arrays, as opposed to three arrays were Node -4 removed. Since Node 3 is a source node for this partition, however, it becomes immediately eligible for clustering with Node 1 in the left-most partition. As a consequence, once transformation is completed, the added cost to achieve loop compatibility, in this case, is the cost of just one array.

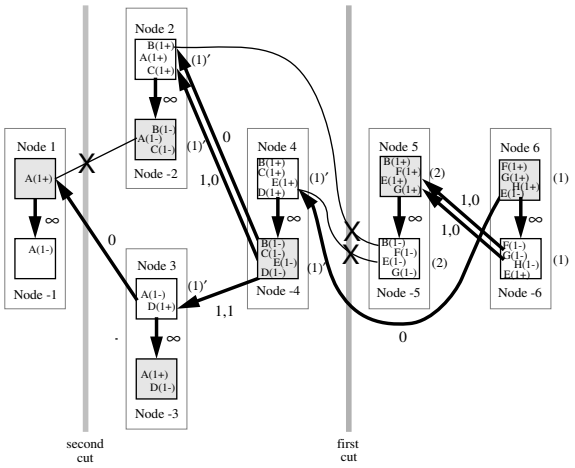
5.4 Analysis of Partitioning Effectiveness

In the previous section we described a method for partitioning a non-compatible cluster of loops into separate fusionable clusters, and in the process we demonstrated the application of this method

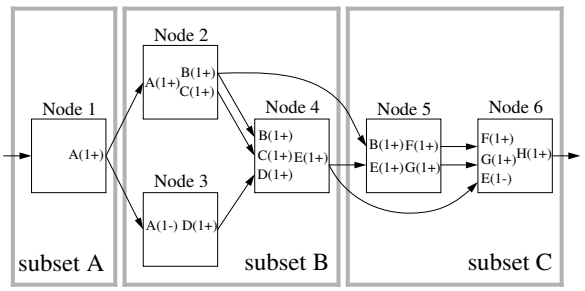
[6] Recall, a constrained loop is one in which there is a loop-carried dependence.

```
      DO 10 I=1, N
10        A(I) = I
      DO 20 I=1, N
          B(I) = A(I) * 2 + 3
20        C(I) = B(I) + 99
      DO 30 I=1, N
30        D(I) = A(N-I+1) + 6
      DO 40 I=1, N
40        E(I) = B(I) + C(I) * D(I)
      DO 50 I=1, N
          F(I) = B(I) * 4 + 2
50        G(I) = E(I) * 8 - 3
      DO 60 I=1, N
60        H(I) = F(I) + G(I) * E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) The Corresponding SG Showing the Partitioning



(c) The Partitioned LCG

```
      DO 10 I=1, N
10        A(I) = I
      DO 20 I=1, N
          b = A(I) * 2 + 3
          B(I) = b
          c = b + 99
          d = A(N-I+1) + 6
20        E(I) = b + c * d
      DO 30 I=1, N
          f = B(I) * 4 + 2
          g = E(I) * 8 - 3
30        H(I) = f + g * E(N-I+1)
```

(d) Code after Loop Fusion and Array Contraction

Figure 16: Multiple-Cut Loop Partitioning

```
FUNCTION partition (graph):  compatible
;; associate a separate colorset with each output node
    source_nodes = getSources(graph)
    FORALL n ∈ source_nodes
        n.colorset = colorset_n
;; color the graph
    compatible = colorTree(source_nodes)
IF NOT compatible
;; create a pseudo-source and pseudo-sink, if necessary
    IF numberOfSources(graph) = 1
        source = source_nodes
    ELSE
        create source
        FORALL n ∈ source_nodes
            attach an arc from source to n
    sink_nodes = getSinks(graph)
    IF numberOfSinks(graph) > 1
        create sink
        FORALL n ∈ sink_nodes
            attach an arc from n to sink
            IF n.node_capacity > 0
                X_nsink.arc_capacity = n.node_capacity
;; partition the graph
    subgraph = flowAugmentingPath(source)
RETURN partition(subgraph)
```

Notice that the `colorTree` function sets a global variable in the event of a loop-carried dependence constraint. To partition the graph when such situations occur requires a modification to the algorithm, which we describe through an illustration later in this section. Notice too that as long as the tail-end subgraph of the SG is non-compatible, the algorithm partitions the graph into increasingly smaller subgraphs. This is accomplished by recursive application of the `partition` function applied to the subgraph on the sink side the SG. As a result, an original cluster of loops might be partitioned several times, each partition resulting in at least one additional fusionable cluster of loops being found.

We examine these processes further in our next example, shown in Figure 16. Besides multiple partitioning, this example illustrates a couple of other unique situations which have not been addressed yet, specifically, the handling of multiple edges between nodes, and the treatment of adjacent cycles when multiple cuts are involved. The FORTRAN loop cluster used in this example is shown in Figure 16(a), prior to loop transformation, and the corresponding IG with its induced SG are shown in Figure 16(b). Examination of the figure reveals that the original loops are non-compatible because there exists three odd-length cycles in the IG; the first involving Blocks 4–6; the second, Blocks 2, 4, and 5; and the third, Blocks 1–4.

A `checkset` which contains the current nodes being visited during breadth-first traversal, and a `nextset` which contains the next level of nodes to visit,

A set of `noncompatibles` = $\emptyset$, which contains any edges identified as having a non-compatible array-reference order with that of its neighbor node; we refer to these edges as X-edges,

A global Boolean value `constrained`, initialized to `false`, indicating non-compatibility because of loop-carried dependence constraints, and

A Boolean return value, `compatible`, with an initial value of `true`.

output: The IG partitioned into components, each component representing a compatible loop cluster.

procedure:

FUNCTION `colorTree (checkset): compatible`
FORALL $n \in$ `checkset`
  FORALL input arcs X_{nm} and the arc to complement node $\overline{n}$
    IF the terminus m is not colored
      color m the opposite color of n
      set m.`node_capacity` = 0
    IF m.`colorset` $\neq$ n.`colorset`
    ;; establish a colorset correspondence
      alias m.`colorset` and n.`colorset`
    IF m is the opposite color of n
    ;; set node and arc capacities
      increment m.`node_capacity` by n.`node_capacity`
      IF m has multiple output ports
        X_{nm}.`arc_capacity` = n.`node_capacity`
      ELSE
        X_{nm}.`arc_capacity` = ∞
      add m to `nextset`
    ELSE IF m is the same color as n
    ;; cut the non-compatible edge
      m.`node_capacity` = X_{nm}.`arc_capacity` = 0
      `compatible` = `false`
    IF m is constrained *;; i.e., it contains a loop-carried dependence*
      IF `color_preference` has not been set
        set `color_preference` to the color of m
      ELSE IF the color of $m \neq$ `color_preference`
        `constrained` = `true`
  IF the `nextset` is not empty
    `compatible` = `colorTree`(`nextset`)
RETURN `compatible`

adjacent cycles, and *2)* loosely connected nodes.[5]

The adjacent cycles are those depicted in the upper-left portion of the figure, $\{1, 2, -3\}$, $\{2, -3, 3, 7, 4\}$, and $\{1, 2, 4, 7, -7, 5\}$. The primary characteristic of such cycles is that the fork and join of one cycle is somehow interleaved with the fork and join of another. In situations of this type, it is necessary to ensure sufficient flow reaches the farthest cycle so that should this cycle need to be cut, it will be cut at its least-cost location—the location corresponding to a fork in the LCG. In this instance, since cycle $\{2, -3, 3, 7, 4\}$ is odd-length, it must be cut, but doing so forces the later cycles, $\{1, 2, -3\}$, and $\{1, 2, 4, 7, -7, 5\}$, to also be cut, even though these later cycles are both of even length. Notice how the node capacities ensure that all of the cuts occur at the correct location. Notice too that this property holds, regardless of whether other subgraphs of intervening chains or even-length cycles are added to these cycles, for example, imagine additional nodes being inserted along arc $(-3, 1)$.

Loosely connected nodes are those which are not a part of some outer-level cycle, such as Node 6. Again, as in the previous situation, node capacity is required to ensure cuts are made at the correct location. Specifically, had node capacities not been used, arc $\{-7, 6\}$ would have had the same capacity as arc $\{6, \texttt{sink}\}$, and as a result, the flow-augmenting path algorithm would have incorrectly cut arc $\{-7, 6\}$, isolating Node 6, instead of making the least-cost cut of the arc to the pseudo sink, as shown. In both this instance and the previous one, adherence to the capacity-weighting scheme described in Item 5, led to optimal partitioning of the graph; whereas, had capacities been assigned less discriminately, optimality would not have been achieved.

5.3 The Partitioning Method and Refinements

In the last section we provided an intuitive description of how loop-cluster partitioning might be accomplished. In this section we formalize the method and provide additional examples illustrating the application of cluster partitioning to special cases involving multiple partitions and then constrained loops. First, we begin with a formal description of the algorithm:

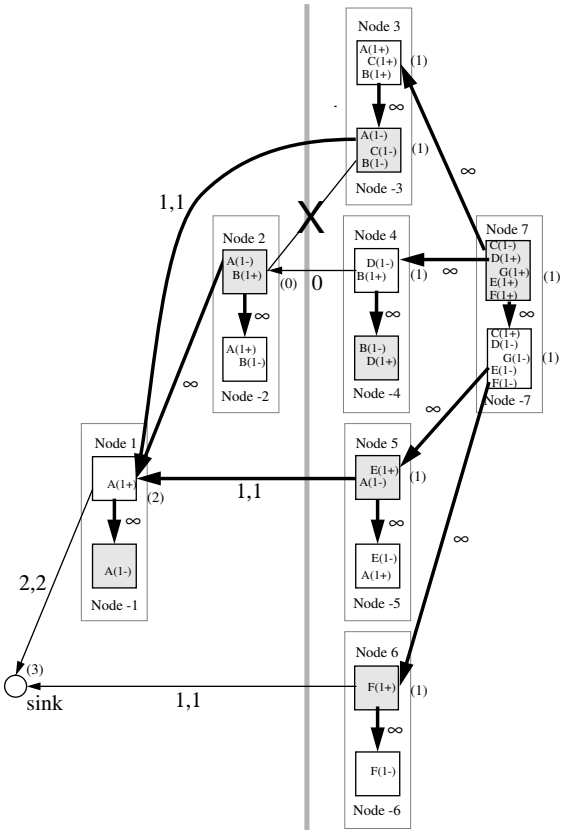
Algorithm III *Partition a cluster of non-compatible loops.*

input: An interference graph, IG = $[[N, \overline{N}], E]$, constructed using Algorithm I (see page*input:* 9).

A $\texttt{colorset}_c = \{\texttt{color}_1, \texttt{color}_2\}$ for each output node in the LCG,

A `color_preference` which determines the color of the set to be chosen in the event the loop cluster contains constrained nodes; initially `color_preference` is undefined,

[5] We are more precise with these terms in Section 5.4.



(a) Partitioned Spanning Graph

```
      DO 10 I=1, N
10        A(I) = 36
      DO 20 I=1, N
20        B(I) = A(N-I+1) + 99
      DO 30 I=1, N
30        C(I) = A(I) + B(I)
      DO 40 I=1, N
40        D(N-I+1) = B(I) * 8 + 2
      DO 50 I=1, N
50        E(I) = A(N-I+1) / 4
      DO 60 I=1, N
60        F(I) = N
      DO 70 I=1, N
70        G(I) = C(N-I+1) * D(I) + E(I) * F(I)
```

(b) Original FORTRAN Code

```
      DO 10 I=1, N
          A(N-I+I) = 36
10        B(I) = A(N-I+1) + 99
      DO 20 I=1, N
          c = A(N-I+1) + B(N-I+1)
          d = B(I) * 8 + 2
          e = A(N-I+1) / 4
          f = N
20        G(I) = c * d + e * f
```

(c) Optimal Solution

Figure 15: Optimal Partitioning using Node Capacities

Let us return to the problem shown in Figure 14 to see how the above strategy is applied. Either during the graph-coloring process or immediately following it, pseudo source nodes and/or pseudo sink nodes are added to the graph, if either are necessary. In this instance, since the SG contains two leaf blocks, Blocks 1 and 2, a sink node is created with arcs extending from the primary nodes of these blocks to this sink.

The compiler next assigns arc capacities, as described in Step 5. For the purpose of clarity, we describe how these capacities are derived rather than enumerating the steps taken during their derivation; the exact procedure is described in the next section. Edge (3,−4) from the IG in Figure 14 is an X-edge and therefore, it is not included in the SG; arc $(5, 3)$, the adjacent arc at the join in the IG at Node 3 is assigned a zero capacity. The second join at Node 2 is part of the even-length fundamental cycle between Blocks 2, 3, and 5. The capacity of Node -3 is zero, because of the zero capacity at Node 3, hence, it has no effect on the capacity of Node 2. On the other hand, Node 5 receives a unit capacity from Node 6, which increments the capacity of Node 2, and accordingly, arc $(5, 2)$ receives a capacity of 1. Applying the same rational to the sink node, we derive a capacity on the arc $(2, \texttt{sink})$ of 1 also. All remaining arcs in the graph are then given an infinite capacity.

Once these capacities have been assigned, the SG is ready for network flow analysis, and the flow-augmenting path algorithm is applied. In this instance, only one path is open, and it is saturated after only one flow-augmenting path is found, $\{6,-6,5,2,sink\}$. At the point of network saturation the graph becomes effectively partitioned, as shown. Although we are assured of the compatibility of nodes prior to the cut, we cannot be assured of the compatibility of nodes after it, since a single cut is often insufficient to break all odd-length cycles in the graph. Therefore, graph-coloring is applied once more to the remaining (left-most) subgraph. In this instance the subgraph remains unchanged by the ensuing analysis, so partitioning is complete. The LCG, after partitioning, is shown in Figure 14(c). Since the compiler now knows that each of the two partitions are compatible, the corresponding loops in each partition can be fused and array contraction applied, as shown in Figure 14(d). As can be seen from the transformed code, the benefit of partitioning can be significant; in this case, it results in elimination of loop control overhead for three loops and scalar-variable replacement of three of the original six arrays.

Before proceeding with the formal description of our loop partitioning algorithm, let us return to the issue of the necessity for Item 5 of the network transformation description (on page 29). Item 5 gives criteria for assigning arc capacities based upon cumulative earlier capacities. The reason for establishing capacities as prescribed by this item is two-fold: *1)* to ensure sufficient flow reaches all arcs in the graph, especially those incident to distant joins, and *2)* to ensure that arcs incident to joins in the SG are given priority over cuts along the respective paths to these joins. The first reason is based upon the fact that, in the absence of sufficient flow, lower-priority arcs can be starved of capacity (and hence cut) preventing flow from reaching possible distant cuts of higher-priority. On the other hand, the arc capacities themselves serve as a means of establishing priority, that is, a means of differentiating the relative importance of one cut as opposed to another.

The impact of capacity assignments is illustrated in the partitioning problem shown in Figure 15. Figure 15(a) depicts the SG generated from the FORTRAN source code shown in Figure 15(b). There are two situations present in this code which differentiate it from our previous examples: *1)*

notions, the solution technique becomes straight forward: Saturate the network with augmenting flows in order to find a minimum capacity cutset. This cutset, in turn, represents the least-cost partition of the SG that we are seeking.

The initial phase in the partitioning process is to transform the SG in Figure 14(b) into the structure of a flow network, as specified in preceding flow-problem formulation. To do this, we make the following correspondence between the two problems:

1. We use the SG to represent the network, as shown in Figure 14(b).
2. We assume that flows go against the flow-dependence arcs in the IG, i.e., instead of the flow going from, say, Node 2 to Node 5, we assume it flows in the opposite direction, from Node 5 to Node 2, as shown in the figure.
3. Because the flows are reversed, we reverse the network source and sink; for example, we let Node 6 be our source, instead of a sink, and in situations in which there are multiple sources and/or sinks, we extend the network by adding an artificial source or sink, as shown. In the Figure 14(b) the artificial sink has been added to the SG to connect the input nodes, 1 and 2. Extending the graph, as we have done, is the usual way to handle multiple sources and sinks; however, instead of assigning infinite capacity to these pseudo-arcs as customarily done, we sometimes assign a lesser capacity, as described next.
4. We assign a zero capacity to each input arc of a join in the SG (or conversely, to each fork in the IG) if the input arc of the SG is adjacent to an X-edge in the IG. For example IG edge $(3, -4)$ is marked with an X, so the adjacent SG input arc $(5, 3)$ is given a capacity of zero.
5. Next we associate with each node a temporary variable, called *node_capacity*, and for each output node of the IG, we initialize the value of this variable to 1; each remaining node capacity is initialized to zero. Then at each terminal node encountered during the breadth-first traversal of the IG (used to create the SG), we add the capacity of the origin node to the current node's capacity, provided the corresponding edge is neither an X-edge or an adjacent zero-weighted arc. At each "compatible" join encountered, i.e., for each join node in the SG that does not have an X-edge or zero-weighted input arc, we assign the arc's origin node capacity to the corresponding arc, if the corresponding origin node is non-zero. In Figure 14(b) for example, Node 3 has zero capacity; Node 2, a capacity of 1 (taken from Node 5); and the sink node, a capacity of 1 also (taken from Node 2).
6. And lastly, we add infinite capacities to all remaining arcs, to ensure that none are cut.

Item 4 is done to ensure that every odd-length cycle is cut, although other higher-priority cuts can, and often do, occur closer to the root(s) of the SG which supersede these "later" cuts. As a result, these later cuts might not actually ever be effected. Ultimately it is the flow-augmenting path algorithm that determines whether these cuts are actually used to partition the graph.

The motivation for Item 5 might be unclear at this point; however, we explain the rational behind it after this next example. For the moment, the underlying notion is merely that the capacities must reflect the relative priority with which cuts are to be made.

cut; otherwise, the loop cluster will not be separated. As a further condition, each cluster created as a result of the partition must itself be a *fusionable partition*. By a fusionable partition we mean that the loops comprising the partition must be compatible in order that they be later combined into a single loop, by loop fusion. Furthermore, we would like the edge-cut that accomplishes the partitioning to be least cost in terms of the number of edges cut. To achieve these objectives, we expand our previous methodology to incorporate a graph-partitioning technique based upon the well-known *Max-Flow Min-Cut* theorem [16].

Theorem 5.3 (Max-Flow Min-Cut) *The maximum value of an (s,t) flow is equal to the minimum capacity (s,t)-cutset.*

The s in the above theorem represents the source of flow, and the t, the sink. For the moment, one might view the "flow" as the pipelined transfer of array elements from one code block to the next.[4] Accordingly, an (s,t) flow is just the flow between the source and the sink, and an (s,t)-cutset is any partition of the network that separates the nodes into two sets, one set containing the source and the other, the sink.

To find the max-flow and min-cut through a network, a series of *flow-augmenting paths* are successively added together until the network becomes saturated. At the point of saturation, network flow becomes maximal, and as Theorem 5.3 suggests, the flow at this point is equal to the flow across the minimum capacity cutset. This property is formally stated in the following related theorem:

Theorem 5.4 *A flow is maximal if and only if it admits no augmenting path from s to t.*

The above theorem assumes several basic network properties apply, based upon the so-called *conservation law*. The conservation law states that the flow into a node must equal the flow leaving it, or

$$\sum_j x_{ji} = \sum_j x_{ij} \tag{1}$$

In the above equation x_{ij} is the flow through arc (i,j). When the preceding relationship is extended to the entire network we have

$$\sum_j x_{ji} - \sum_j x_{ij} = \begin{cases} -v, i = s \\ 0, i \neq s,t \\ v, i = t \end{cases} \tag{2}$$

where v is the value of the flow. Besides flow conservation, on each arc we also have a *capacity constraint*:

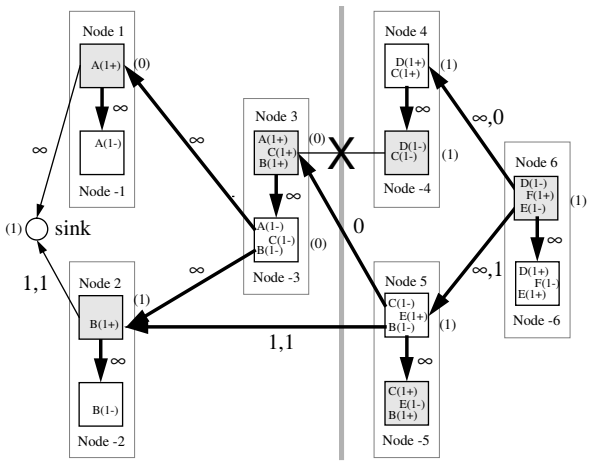
$$0 \leq x_{ij} \leq c_{ij} \tag{3}$$

c_{ij} is the flow capacity of the arc. Any flow satisfying condition 2 and condition 3 is called a *feasible flow*. Lastly, if P is an undirected path from s to t, an arc $(i,j) \in P$ is a *forward* arc if it is directed from s to t; otherwise, it is a *backward* arc. P is said to be a flow-augmenting path with respect to flow x_{ij} if $x_{ij} < c_{ij}$ for each forward arc, and $x_{ij} > 0$ for each backward arc. When we apply these

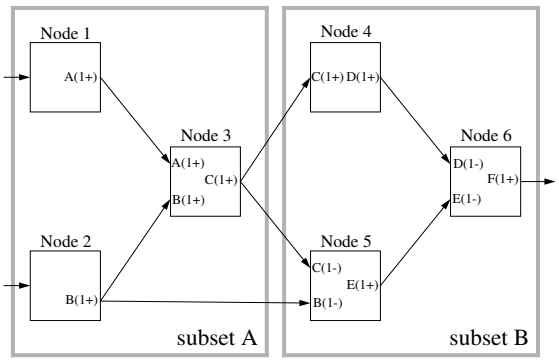
[4] Later we will change this definition slightly to suit our specific purpose.

```
      DO 10 I = 1, N
10        A(I) = I
      DO 20 I = 1, N
20        B(I) = N - I
      DO 30 I = 1, N
30        C(I) = A(I) + B(I)
      DO 40 I = 1, N
40        D(I) = C(I) + 99
      DO 50 I = 1, N
50        E(I) = C(N-I+1) + B(N-I+1)
      DO 60 I = 1, N
60        F(I) = D(N-I+1) + E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) Flow Analysis of the Spanning Graph



c. The Partitioned LCG

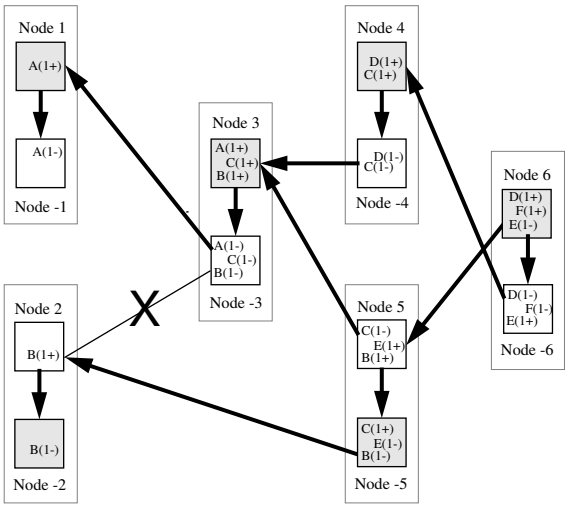
```
      DO 10 I = 1, N
         a = I
         B(I) = N - I
         C(I) = a + B(I)
10    CONTINUE
      DO 20 I = 1, N
         d = C(N-I+1) + 99
         e = C(I) + B(I)
         F(I) = d + e
20    CONTINUE
```

(d) Code after Loop Fusion and Array Contraction

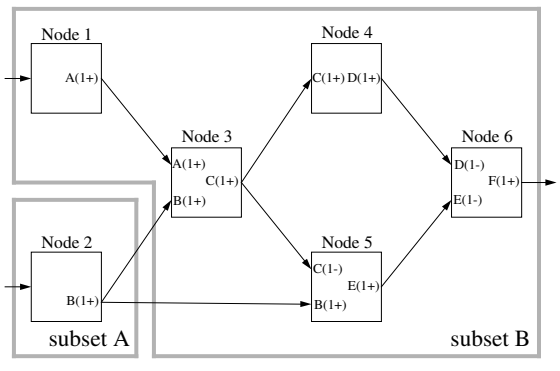
Figure 14: Another Case of Non-Compatible Loops

```
      DO 10 I = 1, N
10        A(I) = I
      DO 20 I = 1, N
20        B(I) = N - I
      DO 30 I = 1, N
30        C(I) = A(I) + B(I)
      DO 40 I = 1, N
40        D(I) = C(I) + 99
      DO 50 I = 1, N
50        E(I) = C(N-I+1) + B(I)
      DO 60 I = 1, N
60        F(I) = D(I) + E(N-I+1)
```

(a) A Non-Compatible Cluster of Loops



(b) The Spanning Graph (SG) Induced from the IG



(c) The Partitioned LCG

```
      DO 10 I = 1, N
         B(I) = N - I
10    CONTINUE
      DO 20 I = 1, N
         a = I
         c = a + B(I)
         d = c + 99
         e = c + B(N-I+1)
         F(I) = d + e
20    CONTINUE
```

(d) Code after Loop Fusion and Array Contraction

Figure 13: A More Complex Cluster of Non-Compatible Loops

other hand, the nodes incident to edge (2,−3) in the IG are of the same color and so an arc would not be added. Note we indicate the non-compatibility of edge (2,−3) by an X.

Now suppose we were to add the non-compatible edge (2,−3) to the tree, knowing that this is not allowed. The result would be the creation of the odd-length cycle $\{2, -3, 3, 5, -5, 2\}$. Since this is the cycle preventing compatibility, it is the one that must be partitioned. Here we consider only *fundamental odd-length cycles* in our analysis, because unless all such cycles are broken, non-compatibility will remain. By fundamental cycles we mean *chordless cycles*; therefore, the cycle $\{2, -3, 3, -4, 4, -6, 6, 5, -5, 2\}$ is not a fundamental cycle because of chord $(3, 5)$. Before continuing, let us reemphasize the following points: *1)* at least two edges of the odd-length cycle must be cut to effectively partition the cycle, *2)* the edge-cut must cut both chains extending between the fork and join of the cycle, and *3)* the least costly edges to cut are those which correspond to a fork in the LCG, since usually only a single array is affected. If we apply these notions to this particular problem in Figure 13, we once more find we are able to partition the graph, just as effectively as before (see Figure 13(c)). Moreover, we are able to obtain much the same benefit we were able to obtain from transforming a fully compatible cluster, as is evident from the transformed code in Figure 13(d). Furthermore, this benefit is achieved with only slightly additional analysis.

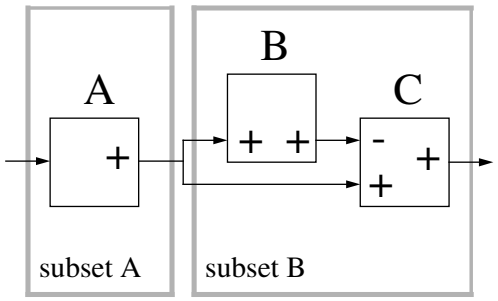
There is one additional assumption we make with respect to the presence of even-length cycles in the SG, e.g., the cycle $\{3, -4, 4, -6, 6, 5, 3\}$ in Figure 13(b). One might rightly ask: How does the addition of these cycles affect the overall analysis? The answer to this question lies in the following theorem:

Theorem 5.2 *The graph formed by inserting an edge between any two nodes of a two-colored graph will itself be two-colored if and only if the nodes incident to the additional edge are opposite in color.*

Proof of the theorem is immediate, by construction. Theorem 5.2 assures us that the even-length cycles have no adverse effect, as far as our analysis is concerned. We might add, however, not only is it safe to insert the compatible edges into the graph, it is essential to do so to ensure completeness. Otherwise, without these edges we would be unable to apply the global analysis needed to obtain minimum-cost partitioning in more complex situations. The significance of this last statement will become clear as our analysis continues.

So far we have only encountered simple cases in which there was little interaction between an odd-length cycle and the rest of the graph. Unfortunately, the situation is often more complicated, and cutting just a single arc, as we did so far, is not enough to effect complete partitioning. This is case in our next example, shown in Figure 14. As in earlier examples, the original code is in Figure 14(a), and the corresponding SG is in Figure 14(b). Also as with previous examples, we show in Figure 14(b) the breadth-first spanning tree and node coloring, constructed as previously described. Notice, however, this time the odd-length cycle and the even-length cycle have been switched. The odd-length cycle is now the cycle $\{3, -4, 4, 6, 6, 3\}$. As a consequence, the previous strategy to partition the graph no longer works. The reason is that there is now an extra edge that must be dealt with that is outside of the odd-length cycle, edge $(2, 5)$.

From our previous examples we know that edges $(3, 4)$ and $(3, 5)$ must be cut to effect the most efficient edge-cut of the odd-length cycle. It is also clear from the figure that edge $(2, 5)$ must be




```
DO I = 1, N
   A(I) = I
END DO
DO I = 1,N
   B(N-I+1) = A(N-I+1)
   C(I) = A(I) + B(N-I+1)
END DO
```

(a) The Loop Communication Graph **(b) Code after Loop Fusion**

Figure 12: A Partition of Three Non-Compatible Loops

partition the cycle,

$$C(5,2) = \frac{5!}{3! \cdot 2!} = 10$$

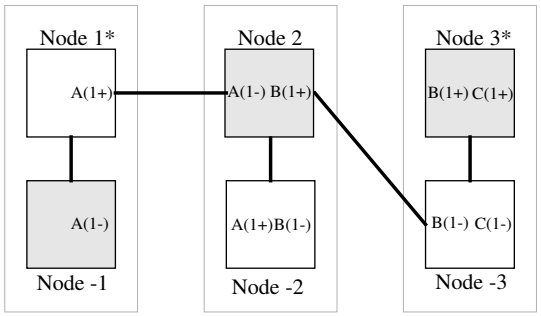
however, not all of these combinations are valid, and others are more costly than the rest. For example, a cut between Node 2 and Node -2 would not be valid since putting these nodes in separate partitions has no meaning in the context of the actual loop cluster. Likewise, a partition such as $\{(1,-2),(2,3)\}$ would not be valid either, since the corresponding loop organization is not programmable. Consequently $2! \cdot 1! = 2$ combinations remain: $\{(1,-2),(1,-3)\}$ and $\{(2,3),(1,-3)\}$. Of these two alternatives, the first is least costly since it affects the fewest number of arrays overall, in this case, only array `A`; whereas the second edge-cut, $\{(2,3),(1,-3)\}$, affects two arrays, `A` and `B`.[3] Note that by cutting an edge we imply the need to pass corresponding array values between code blocks. If we partition the cluster using the first edge-cut, the result would be as shown in Figure 12. In Figure 12(a) we show the partitioned LCG, and in Figure 12(b) we show the resulting code after fusion. Although a simple case, this example introduces the intuition behind the partitioning strategy we describe next.

With this next example, we establish a general framework for solving the cluster partitioning problem, which we will build upon in later examples. The FORTRAN code for this problem, listed in Figure 13(a), is similar to the code in Section 2 used to verify compatibility of non-nested loops (cf., Figure 1, page 7), except the loops in this instance are non-compatible. The non-compatibility of these loops would be discovered during initial graph-coloring analysis. Recall that, during this analysis, a breadth-first spanning tree is constructed and colored, as shown by the bold edges and shaded nodes in the IG, in Figure 13(b). As a part of the process, remaining edges, i.e., those that are not already part of the tree, are checked, and any that can be added, without violating the two-coloring constraint, are added. It is at this point the IG becomes a Spanning Graph, or SG. Arc $(-4,3)$ is added to the SG because the nodes incident to it are each of a different color. On the

[3] One might easily verify that the same observations apply with regard to the simplest example, shown in Figure 9, page 21.

```
DO I = 1, N
      A(I) = A(I-1) + 10
END DO
DO I = 1,N
      B(I) = A(N-I+1)
END DO
DO I = 1, N
      C(I) = C(I-1) + B(N-I+1)
END DO
```

(a) Non-Compatible Loop-Carried Dependence



(b) An Interference Graph Which Is Not 2-Colorable

Figure 11: A Non-Compatible Chain of Loops

A spanning graph is the two-colored spanning tree induced from the IG, with compatible edges from the IG added as arcs and non-compatible edges marked as X-edges. During analysis, the SG is decorated with other information used during the partitioning process; details of this information are explained as we go along. Because the SG starts out as an IG, we periodically refer to the two graphs interchangeably, choosing one name or the other depending upon the graph property we wish to emphasize. As a means of distinction, however, we consistently refer to undirected "edges" in the IG, as opposed to directed "arcs" in the SG.

There are two reasons for using the IG for loop partitioning: First, since it will already have been constructed during previous graph-coloring analysis, construction is free. And second, it contains the information needed to perform partitioning, organized in the way that can be efficiently accessed. Of course, this second consideration is most important.

Without question, the primary objective of cluster partitioning is to aggregate clusters of loops in such a way so as to achieve the best performance overall, and we now show that the best way to accomplish this objective is to find a minimum cost edge-cut of the graph. To understand exactly what is implied by minimum cost, one need only examine Figure 10. Notice, as was proven above, non-compatibility occurs if there exists an odd-length cycle in the graph, cf., Figure 10(b). Consequently, to make the nodes in the graph compatible we must break this cycle, and it must be broken in at least two locations; otherwise, there would be no way to fuse the loops within the individual clusters. Furthermore, were we to cut the graph at only one location, there would be no valid corresponding program representation, as one might readily confirm.[2]

For the odd-length cycle in Figure 10(b) there are ten combinations of edge-cuts which will

[2]This is not true in all cases, e.g., certain types of multiprocessor programming, as we will explain in Section 6. It does however apply in all cases of uniprocessor programming—which is the focus of our research.

proof:

if: Assume G is two-chromatic; its node set V might then be partitioned into two sets V_1 and V_2 so that every edge of G joins a point of V_1 and V_2. Therefore, every cycle $v_1, v_2, v_3, \ldots, v_n, v_1$ in G necessarily has its nodes with odd subscripts in V_1 and nodes with even subscripts in V_2. As a result, the length of every cycle is even.

only if: Assume, without loss of generality, that G is connected (otherwise the components would be considered separately). Take any node $v_1 \in V$ and let V_1 consist of v_1 and all points at even distance from v_1, while $V_2 = V - V_1$. Since all the cycles of G are even, every line of G joins a node of V_1 with a node of V_2. Now suppose there is a line uv that joins two nodes of V_1. The union of shortest paths from v_1 to v and from v_1 to u together with the line uv contains an odd cycle, a contradiction. □

Obviously the odd cycle in the IG in Figures 9 and 10 must be eliminated without altering the meaning of the original code, if transformation is to be possible. Unfortunately, non-compatibilities which are inherent in the original code cannot be eliminated by further direction transformations. Even so, their adverse performance effects can be partially alleviated by breaking the original non-compatible cluster into smaller clusters, each of which is compatible. The loops in each of these subclusters can then be fused and transformations, such as array contraction, performed on the resulting fused loops, as described in Section 3 (page 13).

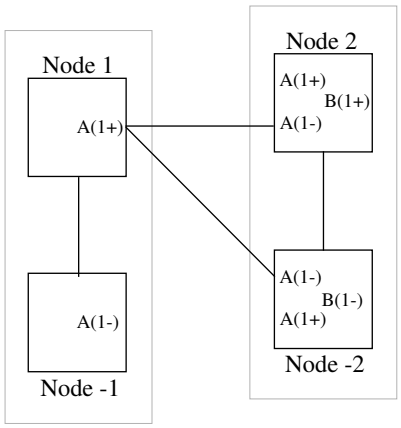
There is one additional source of compatibility we have yet to discuss, and that is the non-compatibility which sometimes occurs among loops in a cluster as a result of conflicting reference-order constraints caused by loop-carried dependence. An example of just such a situation is shown in Figure 11. Here the source of loop non-compatibility is the conflicting array-reference orders imposed on the first and third loops by the loop-carried dependence within each loop, cf. Figure 11(a). In Figure 11(b) we show the corresponding IG as it would appear following compatibility analysis. Notice that in this graph the two non-compatible nodes, Nodes 1 and 3, are separated by a non-compatible (odd-length) chain of nodes. From our previous analysis we know this chain is non-compatible because the two end nodes in the IG (Nodes 1 and 3) are of opposite color. All that would be needed to make this chain compatible would be to remove the loop-carried dependence from either of the two end nodes—but of course, there is no way of doing this. Consequently, there is no remedy but to remove an edge from the chain, in effect, separating the constrained nodes by putting them in different partitions, as in the previous situations. There are usually several alternatives for accomplishing such a partitioning, and after we have laid additional groundwork, we return to this problem to see how partitioning might be accomplished.

5.2 Properties of Non-Compatible Clusters

In this section we continue our analysis of non-compatible loops. In the process we isolate additional properties, besides those already mentioned, which affect partitioning effectiveness. In the course of this investigation we establish a number of guidelines which ultimately form the basis of the partitioning algorithm we later describe. Before we begin, however, a comment about the graphical construct we use during partitioning analysis: the Spanning Graph (SG).

```
DO I = 1, N
   A(I) = I
END DO
DO I = 1, N
   B(I) = A(I) + A(N-I+1)
END DO
```

(a) Non-Compatible Loop Directions among Two Loops

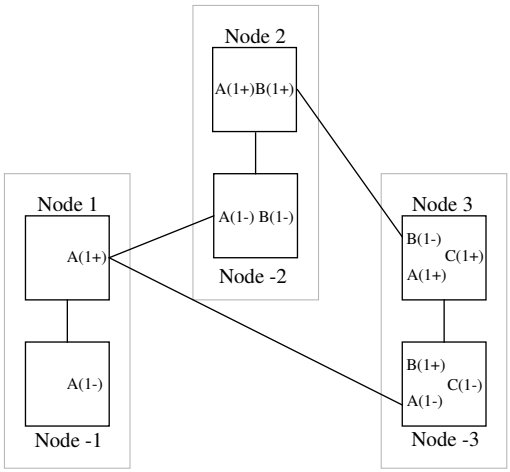


(b) An Interference Graph Which Is Not 2-Colorable

Figure 9: Two Non-Compatible Loops

```
DO I = 1, N
     A(I) = I
END DO
DO I = 1,N
     B(I) = A(I)
END DO
DO I = 1, N
     C(I) = A(I) + B(N-I+1)
END DO
```

(a) Non-Compatible Directions among Multiple Loops



(b) An Interference Graph Which Is Not 2-Colorable

Figure 10: A Cluster of Three Non-Compatible Loops

loops are already compatible in the sense of our Case-1 transformation above. In other words, loop interchange is unnecessary to make the cluster compatible. In this instance, only the direction of the AVs in the various code blocks are allowed to vary. In this special case, analysis proceeds exactly as it did for Case 2 with a separate phase of analysis for each element of the AVs within the cluster.

5 Non-Compatible Loop Clusters

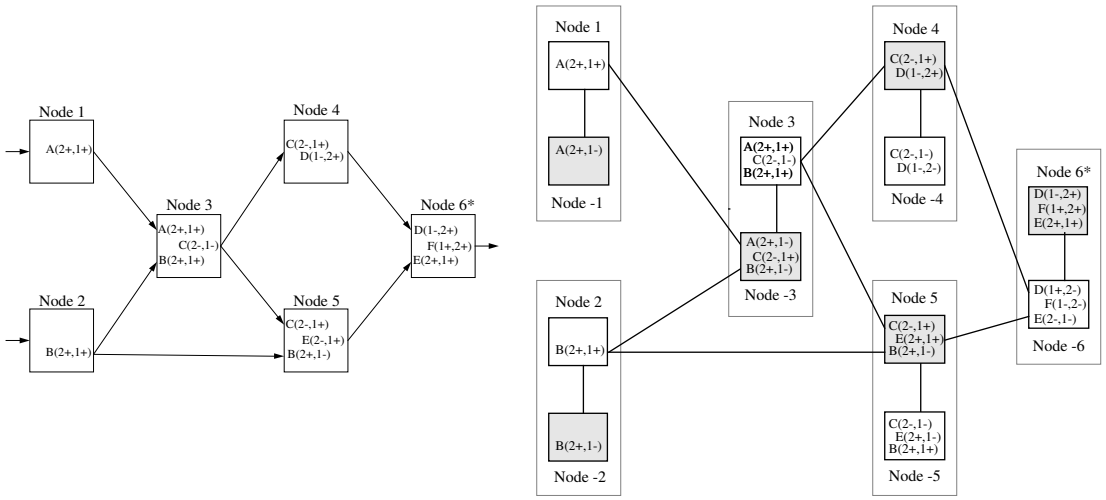
Not all loop clusters are able to be transformed into compatible clusters by the graph-coloring technique described in the last section. Nevertheless, we can usually partition a cluster into a set of smaller clusters which are compatible. Moreover, by cluster partitioning, much of the benefit from collective loop analysis can still be obtained. This then is the problem we examine in this section: how to partition a cluster of non-compatible loops efficiently to achieve best performance.

5.1 Sources of Non-Compatibility

Let us first examine the implications of loop non-compatibility. As noted, compatibility does not always exist among the constituent loops of a cluster. Therefore, as a first step, we would like to be able to identify the conditions which are at the source of the compatibility problem. As we will show, this information is important to being able to partition loops efficiently, both from the standpoint of the transformation itself, as well as from the standpoint of the performance of the resulting code. We begin our examination of non-compatibility analysis by illustrating two of the simplest cases. The first is shown in Figure 9. In this example, array `A` is generated in the first code block and referenced twice in the body of the second, each set of references being made in a different sequence, due to a different loop direction—the first set of references, in ascending order, and the second, in descending order. Observe that, in Figure 9(b), the IG is not two-colorable, thus providing graphic evidence of the non-compatibility of the two loops. Also, observe that the non-compatibility originates at the location of a *fork* in the LCG. For ease of expressibility, we refer to such locations as forks in the IG, as well.

Next let us complicate the situation by adding a third code block to a cluster, as shown in Figure 10. Notice that in Figure 10(b) the generation of array `A` in Block 1 constrains the allowable reference sequences of `A` in Blocks 2 and 3. Consequently, for the last two code blocks to become compatible with the first, array `A` must be generated in both ascending and descending sequences simultaneously, an impossibility. Notice again, our previous observation once again applies: non-compatibility originates at the location of a fork—also notice that the graph is not two-colorable. The fact that neither this graph nor the previous one is two-colorable, hence, the fact that we are unable to find a compatible set of loop directions, is readily explained by a fundamental theorem from graph theory [5, 8, 14]:

Theorem 5.1 *A graph with at least one edge is two-chromatic if and only if it has no cycles of odd length.*



(a) LCG prior to Inner-Loop Reversal **(b) IG used for Second Case-2 Transformation**

Figure 8: Phase-3 Analysis of a Two-Dimensional Loop

When loop fusion is applied, the original loop cluster is transformed into the single doubly-nested loop shown below, to the left, and once this is done, all of the array references except the last (the reference to the output array `F`) are replaced by corresponding scalar variables, as shown on the right.

After Loop Fusion

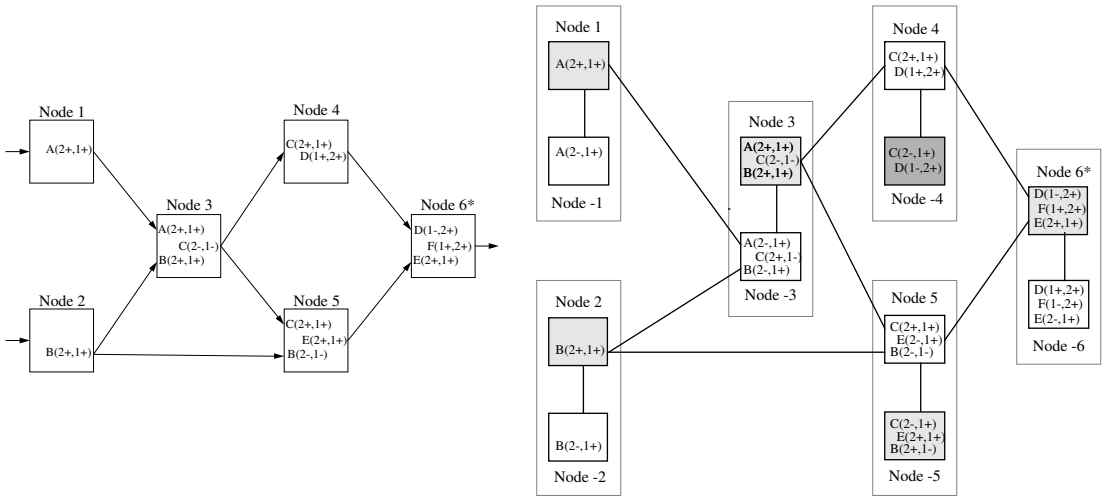
```
DO I = 1, N
   DO J = 1, N
      A(J,N-I+1) = (N-I+1) * J
      B(J,N-I+1) = 1 - I
      C(N-J+1,I) = A(J,N-I+1) + B(J,N-I+1)
      D(N-I+1,J) = C(N-J+1,I) + 99
      E(J,I) = C(N-J+1,I) + B(J,N-I+1)
      F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
   END DO
END DO
```

After Array Contraction

```
DO I = 1, N
   DO J = 1, N
      a = (N-I+1) * J
      b = 1 - I
      c = a + b
      d = c + 99
      e = c + b
      F(I,J) = F(I-1,J) + d + e
   END DO
END DO
```

Note that in this case, the savings in storage and memory bandwidth is $5 \cdot n^2$, where n is the number of loops in the cluster. Again, in addition there are other advantages derived from increasing code-block size, advantages cited earlier for the non-nested case (see Section 3).

When the notions of the previous section are extended to the multiple nestings, the problem quickly becomes intractable because of the rapid growth of possibilities. Nonetheless, one special case does warrant mentioning; this is the extension of Case 2 from the previous section to the general case in which nesting of loops among the various code blocks comprising the cluster of



(a) LCG prior to Outer-Loop Reversal **(b) IG used for First Case-2 Transformation**

Figure 7: Phase-2 Analysis of a Two-Dimensional Loop

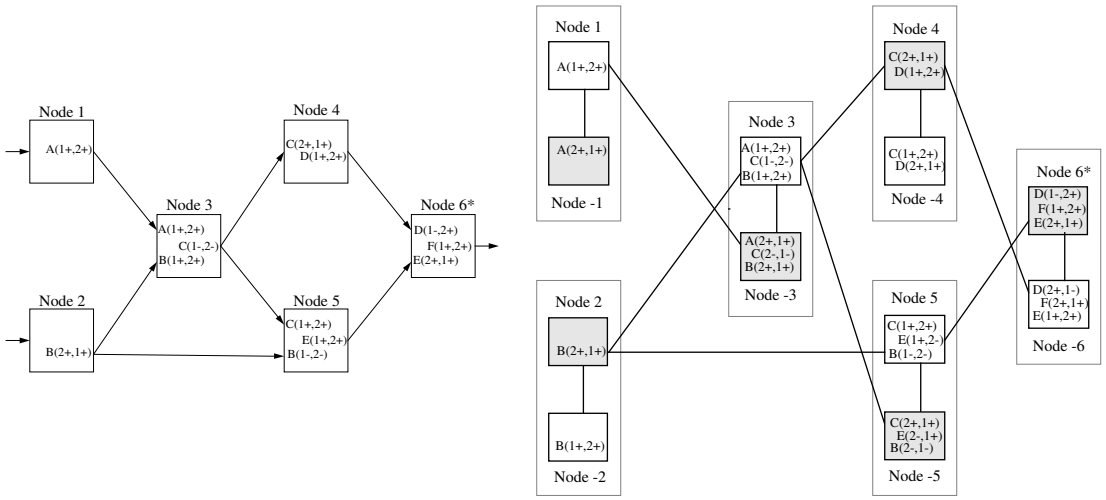
on to the third and final phase of analysis.

Phase 3 is just Phase 2 performed on the inside loop of each node in the cluster rather than on the outside node. Using the output from the previous phase, i.e., the LCG shown in Figure 8(a), the compiler creates the IG shown in Figure 8(b). This time the complementary nodes are created by reversing the direction of the last element of each AV of a corresponding primary node. Phase-3 analysis is also successful, leaving the nodes in the shaded boxes in the IG in Figure 8(b) as the final set of nodes from the overall loop pipelining transformation. Again in this last phase, as in the previous one, the transformation required to bring the cluster into alignment is loop reversal.

Now that a compatible set of array reference orders is known, it is a straight-forward process to fuse statements into a single code block and then eliminate the intermediate arrays by array contraction, as one done in the case of non-nested loops. To demonstrate this, we continue with the example, using the doubly-nested loops indicated by the shaded boxes in Figure 8(b). The retransformed code based upon the preceding analysis is shown next:

After Direction Reversal

```
      DO 10 I = 1, N
         DO 10 J = 1, N
10          A(J,N-I+I) = (N-I+1) * J
      DO 20 I = 1, N
         DO 20 J = 1, N
20          B(J,N-I+1) = N - (N-I+1)
      DO 30 I = 1, N
         DO 30 J = 1, N
30          C(N-J+1,I) = A(J,N-I+1) + B(J,N-I+1)
      DO 40 I = 1, N
         DO 40 J = 1, N
40          D(N-I+1,J) = C(N-J+1,I) + 99
      DO 50 I = 1, N
         DO 50 J = 1, N
50          E(J,I) = C(N-J+1,I) + B(J,N-I+1)
      DO 60 I = 1, N
         DO 60 J = 1, N
60          F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
```



(a) LCG prior to Interchange **(b) IG used for Case-1**

Figure 6: Phase-1 Analysis of a Two-Dimensional Loop

This last situation is a combination of Case 1 and Case 2. To analyze loop clusters of this type, we first attempt to color the transformed LCG based upon subscript order, as described in Case 1, and if the coloring is successful, we continue by attempting to color the LCG based upon loop directions, as described in Case 2.

An example of the combined approach is shown in Figures 5–8. The cluster of doubly-nested loops used in this example are based upon the code segment shown in Figure 5. The corresponding LCG for this code is shown in Figure 6(a). The loop analyzer of the compiler uses this LCG during the first phase of its analysis to create the IG shown in Figure 6(b). Phase-1 analysis is merely the Case-1 analysis, described above. Notice that since the compiler's first attempt at transformation involves loop interchange, the complementary nodes in the Phase-1 IG are composed by copying the primary nodes, which were taken from the LCG, and reversing the of order of their AV elements (as opposed to changing the direction of these elements as was done in the case of non-nested loop clusters). The result of this analysis, a two-coloring of the IG, is also reflected in Figure 6(b). Although the coloring produces two sets of nodes: those consisting of shaded boxes and those without, only the first is permissible because Node 6 is constrained by the loop-carried dependence which affects array `F`. Thus this set of shaded nodes forms the LCG used during the second phase of analysis. The LCG which depicts the results of Phase-1 analysis is shown in Figure 7(a).

Since Phase-1 analysis uncovered a compatible arrangement of nestings among the entire cluster, the compiler can proceed to the next phase of analysis. This second phase and the one that follows together form the Case-2 situation described above. During Phase-2 analysis, focus is on finding a compatible set of loop directions for each of the outside loops of the code blocks represented by the nodes of the transformed LCG. Therefore, the complementary nodes of the IG in this case are a copy of the primary nodes from Figure 7(a), with the direction of the first element of each AV reversed. The IG reflecting this set of alternative sequences and the resulting two-coloring is shown in Figure 7(b). Fortunately, Phase-2 analysis is successful too, so the compiler is able to proceed

```
      DO 10 I = 1, N
         DO 10 J = 1, N
10          A(I,J) = I * J
      DO 20 I = 1, N
         DO 20 J = 1, N
20          B(J,I) = N - I
      DO 30 I = 1, N
         DO 30 J = 1, N
30          C(N-I+1,N-J+1) = A(I,J) + B(I,J)
      DO 40 I = 1, N
         DO 40 J = 1, N
40          D(I,J) = C(J,I) + 99
      DO 50 I = 1, N
         DO 50 J = 1, N
50          E(I,J) = C(I,J) + B(N-I+1,N-J+1)
      DO 60 I = 1, N
         DO 60 J = 1, N
60          F(I,J) = F(I-1,J) + D(N-I+1,J) + E(J,I)
```

Figure 5: A Collection of Typical Two-Dimensional FORTRAN Loops

In this situation, AVs for the arrays referenced within the cluster are either of the form *array_name(1+,2+)* or *array_name(2+,1+)* or they are of the form *array_name(1−,2−)* or *array_name(2−,1−)*. In either situation, the direction of the loops are presumed to be compatible (so no direction transformation is required). On the other hand, the order of the array subscripts creates the bipartite condition required for two-coloring the IG. In this instance we use Algorithm II, as before, but create the complementary nodes using AVs with a subscript order opposite to the subscript order of the AVs in the primary nodes. Also, instead of using loop reversal, as we did previously, we use loop interchange to align array access orders consistent with the results of the coloring analysis.

CASE 2 *The order of the subscripts are the same across the cluster of loops, but the direction of the loops varies.*

The AVs in this case are either of the form *array_name(1±,2±)* or the form *array_name(2±,1±)*. To analyze these loops, we treat the problem as though it were two separate problems, analyzing the same AV element from each AV as though the referenced arrays were one-dimensional. For example, we might first analyze the directions of the outer loops of the code blocks in the cluster, and if a compatible direction is found, use the results of this analysis to analyze the directions of the inner loops (or visa versa). As in our original examples, the transformation technique used to bring the unaligned loops into alignment is loop reversal.

CASE 3 *Both the order of the subscripts and the direction of the loops vary.*

Array Contraction

```
DO I = 1, N
    a = N - I + 1
    b = I - 1
    c = a + b
    d = c + 99
    e = c + b
    F(I+1) = F(I) + d + e
END DO
```

Notice that array contraction replaced, by a scalar variable, all array references in which an element reference was first produced then later used. For example, `A(N-I+1)` is replaced by `a`, and so forth. Since all but the last array was produced then consumed in an element-by-element manner, the need for intermediate arrays, as in the original code, no longer exists. In later sections we will show that yet additional transformation is not only possible, but, beneficial.

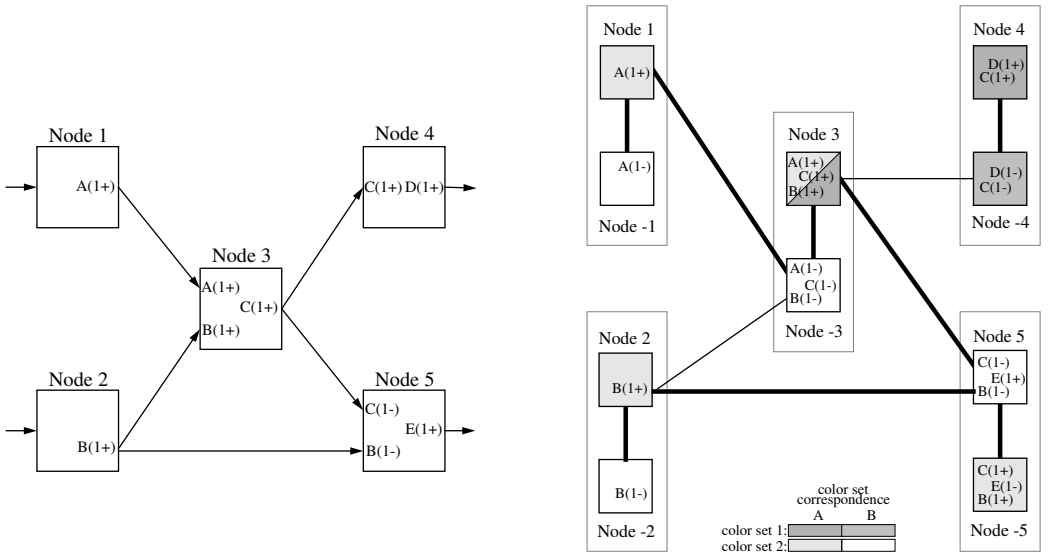
The effect of collective transformations, such as these, can be dramatic; loop overhead and memory requirements can be reduced, and most array indexing operations can be eliminated. Moreover, a single larger code block results which facilitates instruction scheduling.

4 Analysis of Doubly-Nested Loops

So far we have only discussed the situation involving single-nested loops on one-dimensional arrays since this situation is clearly the most intuitive and easiest to handle. The difficulty with multi-nested loops and multi-dimension arrays stems, of course, from the enormous increase in complexity which arises from the many permutations of the levels of loop nesting and loop direction. Whereas in the non-nested case AVs have only two possible orders, in doubly-nested cases there are $2^2 \cdot 2! = 8$ possible sequences. In spite of this enormous increase in complexity, the basic methodology described in the previous section still applies—that is, with a few appropriate enhancements. Recall from the previous section, our strategy was to identify an alternative, or complementary, set of reference orders for each AV in the loop cluster, and then analyze each of the enumerated possibilities using the graph coloring technique described in Algorithm II. In this section we show how this same methodology can be applied to certain situations involving clusters of doubly-nested loops and two-dimensional arrays.

In the case of doubly-nested loops, the compiler faces essentially three possible possibilities: *1)* the direction of all loops are the same across the entire cluster of loops, but the order of the array subscripts in each reference varies, *2)* the order of the subscripts are the same across the entire cluster of loops, but the direction of the corresponding loops in the cluster varies, and *3)* a combination of these two situations in which both the order of the subscripts and the direction of the loops vary across the cluster of loops.

CASE 1 *The direction of the loops are the same across the cluster of loops, but the order of the subscripts varies.*



(a) Loop Communication Graph **(b) Colored Interference Graph**

Figure 4: Loop Collection having Multiple Output Blocks

---

code, shown in Figure 1, results the transformed code shown below, on the left-hand side. Since normalization ensures uniform loop control and since coloring analysis ensures loop compatibility, we know too that the loops in our example cluster can now be safely fused, as shown at the top of the next page, on the right-hand side.

After Direction Reversal

```
DO I = 1, N
   A(N-I+1) = N - I + 1
END DO
DO I = 1, N
   B(N-I+1) = N - (N-I+1)
END DO
C(1:N:-1) = A(1:N:-1) + B(1:N:-1)
D(1:N:-1) = C(1:N:-1) + 99
E(1:N:1) = C(N:1:-1) + B(N:1:-1)
DO I = 1, N
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

After Loop Fusion

```
DO I = 1, N
   A(N-I+1) = N - I + 1
   B(N-I+1) = I - 1
   C(N-I+1) = A(N-I+1) + B(N-I+1)
   D(N-I+1) = C(N-I+1) + 99
   E(I) = C(N-I+1) + B(N-I+1)
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

The final transformation, for now anyway, is array contraction, and the following code is the result of applying contraction to the previously transformed code:

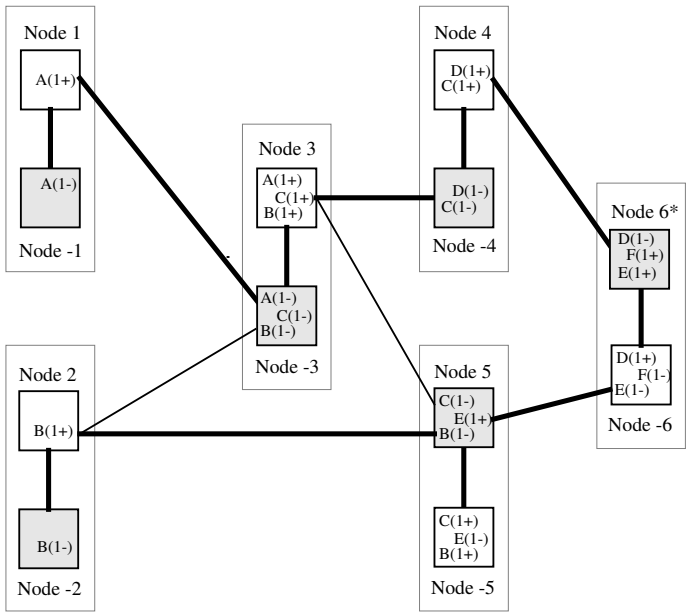


Figure 3: An Interference Graph after Coloring

treated as though they were one. Then, should further traversal lead to some later convergence of the two color sets, at the point of terminus checking from some node, the decision whether to add the new edge, or not, will be based upon the initial color-set equivalence established when the color sets first converged. An illustration of the coloring of a loop cluster with two output code blocks is shown in Figure 4; the LCG is shown in Figure 4(a) and the corresponding IG after coloring is shown in Figure 4(b).

3 Follow-On Transformations

In the last section we described in general terms the collective analysis of the example cluster of loops given in Figure 1 (see page 7). To put this analysis into perspective, we now step through the basic actions taken once a compatible set of loop directions have been found. We begin our follow-on description with the coloring shown in Figure 2 (page 8). Notice that Node 6 is constrained because of a loop-carried dependence of array `F`. As a result, there is only one set of compatible sequences from which to choose, instead of two as would have been the case had the loop-carried dependence of `F` not existed. This set is the one indicated by the shaded nodes in the figure—the cluster consisting of Nodes -1, -2, -3, -4, 5, and 6.

Once the set of compatible nodes have been selected, we apply direction reversal to the loops corresponding to the negative nodes in the set, i.e., the nodes having a negative node number. The basic notion behind direction reversal is merely to substitute `N-I+1` for each occurrence of `I` in the loop body. In this case, applying direction reversal to the first four loops of the normalized

A set of `noncompatibles` = $\emptyset$, which contains any edges identified as having a non-compatible array-reference order with that of its neighbor node (later, we refer to these edges as X-edges), and

A Boolean return value, `compatible`, with an initial value of `true`.

output: An IG with compatible nodes indicated by the primary color in the color set; however, if any nodes are constrained, the compatible nodes will be those with the same color as the `color_preference`.

procedure:

```
FUNCTION colorTree (checkset):  compatible
FORALL n ∈ checkset
    FORALL input arcs X_mn and the arc to complement node n̄
        IF the terminus m is not colored
            color m the opposite color of n
            add m to nextset
        ELSE IF m is the same color as n
            add X_mn to noncompatibles
            compatible = false
        IF m is constrained
            IF color_preference has not been set
                set color_preference to the color of m
            ELSE IF the color of m ≠ color_preference
                add X_mn to noncompatibles
                compatible = false
        IF nextset ≠ ∅
            compatible = colorTree(nextset)
    RETURN compatible
```

The graph in Figure 3 shows the IG in Figure 2 after it has been colored using Algorithm II. The dark edges define the spanning tree created during algorithm execution, and the shaded nodes indicate the set of compatible loops. Note that the constraint on Node 6 prevents the unshaded nodes from also forming a alternative set of compatible directions for the respective loops.

Algorithm II handles situations in which there is only a single output block in the LCG, but situations in which there are multiple output blocks often occur which require a slightly more complex solution, one involving the use of multiple color sets—or rather a different color set for each of the output blocks within the original cluster. In this case, a primary node from each output block is selected to begin the algorithm, and each root node is colored, as in Algorithm II—each with a color from its own respective color set. Then, as before, a depth-first walk is performed beginning from each root. If the LCG is connected (and we can assume that it is), there will come a point where a node is reached that has already been colored, but with a color from a different color set. When this occurs, the corresponding colors from each of the two sets are henceforth

breadth-first walk of the IG, commencing at the output block of a loop cluster. Our particular approach (which is slightly different from the one proposed by Sarkar and Gao) has a special property in that it identifies the least costly non-compatible interference edges to remove whenever no overall set of compatible array-reference orders exits, and it does so efficiently, as we will show in Section 5. The cost to which we refer is the cost of referencing elements of an array from memory, and the least-cost edges are those corresponding to arrays that are used by multiple blocks, not just one. The non-compatible edges, if any are found, ultimately form part of a cut-set used to find compatible clusters of the original cluster which can ultimately be fused, allowing elimination of all arrays except the ones that cross a cluster boundary. We discuss *cluster partitioning*, in detail in the next section.

Again, the basic strategy behind compatibility analysis is to create a two-colored spanning tree during traversal of the IG. We proceed by first identifying and coloring the primary node of the output block of the IG. Then, starting with this node, we traverse the remaining nodes in the graph, in a breadth-first manner. At each node visited during our traversal we examine both the input ports and the port to the complement node within the same block, in each case inspecting the terminus of the interference edge. If the terminus is uncolored, we color it the opposite color of the current node. If, on the other hand, it has already been colored, but the color is the same as the color of the current node, we note the fact that the cluster is non-compatible and store a reference to the non-compatible interference edge to be used during subsequent cluster-partitioning analysis. After all nodes have been reached, the algorithm stops. Nodes that are of the same color represent a compatible set of loops. Of course, this leaves a choice of the two colorings. One guideline regarding which color nodes to choose is to select nodes that correspond to the color of most of the "primary" nodes, since doing so involves the least follow-on direction-reversal (the direction-reversal transformation itself will be explained shortly). On the other hand, choosing a color might not be an option at all if the graph contains constrained nodes, since the color of any constrained nodes will uniquely determine the color set which must be used, that is, of course, assuming all constrained nodes are of the same color. Otherwise, if the constrained nodes are not all of the same color, the cluster will not be compatible overall. Our procedure for testing loop compatibility is summarized in the following recursive algorithm:

Algorithm II *Determine a set of compatible array-reference orders for a cluster of loops, or verify that a compatible set of loops does not exist.*

input: An IG = $[[N, \overline{N}], E]$ in the form of a linked list; each node in the list is either a primary node n^+ or a compliment node n^- (see Figure 2),

A `colorset` = $\{\texttt{color}_1, \texttt{color}_2\}$, and a `color_preference` which determines the color of the set to be chosen in the event the loop cluster contains constrained nodes; initially `color_preference` is undefined,

A `checkset` which initially contains the primary node of the output block of the cluster; we refer to this node as `root`, and `root.color` = $\texttt{color}_1$,

A `nextset` = $\emptyset$, representing the next level of nodes to visit,

restricting analysis to just two orders at a time, say, ascending and/or descending loop direction, problem size is significantly reduced. Moreover, by considering only two alternative orders at a time, we can easily represent any of the 2^n remaining possibilities with a just single IG (again assuming n is the number of AVs). From construction of the IG, we know that neighbor nodes always reflect a complementary array-reference order, and for an overall compatible order to exist, it must consist of orders taken from alternating nodes. Consequently, the IG must be a bipartite graph if a compatible set of loop directions is to exist. Although the number of possibilities can still be very large, we know that there can now be at most two compatible orders for any cluster overall. This fact is what finally makes the problem tractable, since to determine whether a compatible loop ordering exists, we need only determine whether the particular IG under analysis is, in fact, a bipartite graph.

To take advantage of this bipartite property, we look to a few simple notions from graph theory [5, 8, 14], and we begin under the premise, based upon our above observations, that *an IG is bipartite if, and only if, an overall compatible set of array-reference orders exists.* Accepting this fact, it is now easy to show that any bipartite graph is two-colorable (the proof by construction is trivial): Color elements of one partition one color and elements of the remaining partition a second color. Likewise, it follows immediately from the general definition of a spanning tree that the spanning tree induced by an IG represents a minimal connection of nodes within that IG. Therefore our next step to show that every nontrivial (spanning) tree (induced from an IG) is two-colorable.

Theorem 2.1 *Every nontrivial tree is two-colorable.*

proof: (by induction)

basis: Assume T is a tree with two nodes. If one of the nodes is colored with one color, the only remaining node can be colored with a second color.

inductive step: Assume T contains n nodes that have already been successfully colored with two colors. If another node is added, regardless of its location, it can always be colored an opposite color to the color of the node to which it is attached. □

Once a spanning tree is formed within the IG, we are assured of creating a simple cycle for each of the remaining original arcs added to tree. Fortunately determining whether the cycles formed in this way are themselves two-colorable is quite easy; we merely inspect the origin and terminus of each edge being added, and if the two have different colors, edge insertion can be safely accomplished, preserving the colorability of the graph. If edge insertion is successful for all remaining edges, we know the IG overall is two-colorable, thus confirming the fact that the IG is a bipartite graph and that a set of compatible loops exists. Now to find this order we need only select a set of nodes of the same color. This is because the way we constructed the graph assures us of a direct correspondence between compatible loop configurations and node coloring, that is, we know for a compatible cluster of loops, opposite nodes of every interference edge are both an opposite loop configuration and an opposite color.

The preceding analysis suggests how the determination of loop compatibility can be accomplished. The approach we describe is based upon creating a two-colored spanning tree during a

array-reference orders for AVs `A(1+)` in Node 1 and `A(1-)` in Node -3, while no arc was attached between Nodes -1 and 3, even though the non-compatibility of the complement AVs, `A(-)` and `A(+)`, respectively, is equally valid. The absence of "redundant" edges from the complement nodes in each block not only reduces the number of linkages which must be made to the graph, but it also simplifies the ensuing analysis, as will become evident.

Algorithm I *Construct an Interference Graph (IG).*

input: An LCG = $[N, A]$ in the form of a linked list; each node in the list corresponds to a code block. The nodes n are each specified by a set of input and output ports, n_x. Each port refers to an AV for some array X referenced within the block, along with list of connecting nodes m indicating from which node array X was produced or to which m it is being sent.

output: An IG = $[[N, \overline{N}], E]$ created from the LCG; each block in the LCG consists of two nodes: a primary node n^+ containing the same input and output ports as corresponding node n in the LCG and a complement node n^- containing the same input and output ports, but with AV's n_x, $\forall x \in n$, switched to indicate the alternative array-reference order (see Figure 2).

procedure:

Create a complement field in each node.

FORALL $n \in N$ in the LCG
copy n to primary node n^+ and to complement node n^-
FORALL $n_x \in n^-$
switch the direction of AV elements
insert nodes n^+ and n^- into the graph

Add interference edges.

FORALL $n^+ \in N$
FORALL output ports n_x
IF m_x for node m specifies the alternate vector
attach a link to $m^{\pm}$ creating $X_{n+m\pm}$
ELSE
attach a link to $m^{\mp}$, creating $X_{n+m\mp}$

2.3 Justification and Methodology

Were we to consider all possible array-reference orders possible from a cluster of loops, there would be $(k! \cdot 2^k)^n$ such possibilities, where n is the total number of AVs used within the code blocks of the loop cluster and k is the level of nesting of these loops. Obviously, analysis of so many alternatives at a time would be intractable for all but the simplest loop clusters. However, by

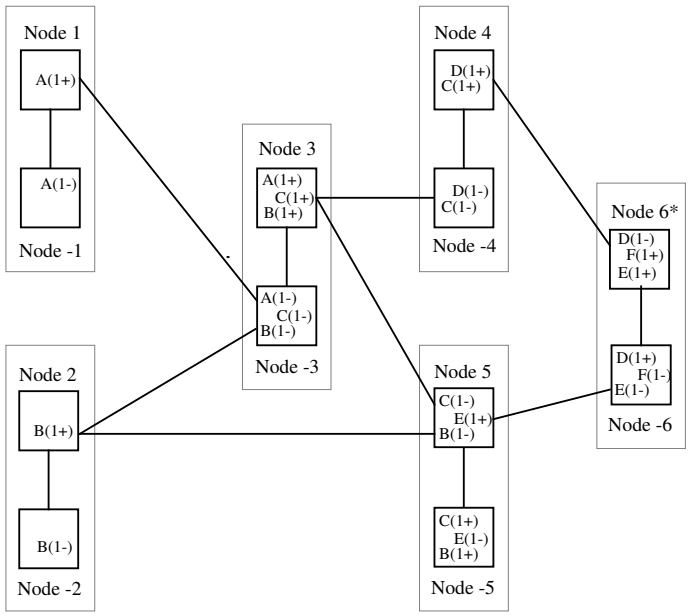


Figure 2: An Interference Graph (IG)

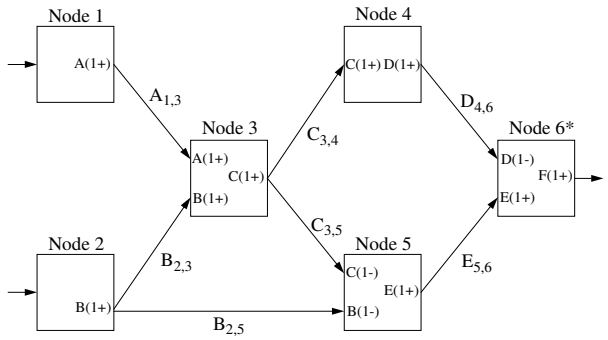
nesting level is used, corresponding elements in the the AV tuple are switched. For example, the first block, corresponding to Node 1 in the LCG, shown in Figure 1(a), consists of two nodes; a primary node n_1^+, labeled "Node 1" with AV `A(1+)`, and directly below it, a compliment node n_1^-, labeled "Node -1" with AV `A(1-)`. Formally, an $IG = [[N, \overline{N}], E : E \in N \times N]$, where $[N, \overline{N}]$ is a block, $n^+ \in N$ and $n^- \in \overline{N}$ are nodes within a block, for $1 \leq n \leq |N|$. An undirected edge $X_{m,n} = (m^+, n^\pm) \in E$, called an *interference edge*, represents non-compatibility between the order of generation (definition) of array values in one code block and the order of their consumption (use) in another.

Obviously a code block cannot simultaneously process array elements in more than one order at a time. Therefore, to reflect the fact that alternative reference orders represented by the primary and compliment nodes within a block are not simultaneously possible, we attach an interference edge between the two. Likewise, we attach an edge between the output port of a node in one block and input port of a node in another when the two ports reference the same array and the corresponding AVs in the two nodes differ. In this second situation, the edge indicates a situation we would prefer to avoid, since unless the AVs are the same, the array elements which pass between the two nodes, or code blocks, cannot be uniformly passed from one block to the other, as we would like.

Note that not all interference edges need to be attached, since the edges extending from the primary nodes to the nodes in follow-on code blocks provide sufficient interblock compatibility information, by themselves, to allow us to do our analysis. Attaching the corresponding arcs from the complement nodes to follow-on blocks therefore becomes unnecessary. For example, notice that, in Figure 2, an edge has been attached between Nodes 1 and -3 to indicate non-compatibility in the

```
DO I = 1, N
   A(I) = I
END DO
DO I = 1, N
   B(I) = N - I
END DO
C(1:N:1) = A(1:N:1) + B(1:N:1)
D(1:N:1) = C(1:N:1) + 99
E(1:N:1) = C(N:1:-1) + B(N:1:-1)
DO I = 1, N
   F(I+1) = F(I) + D(N-I+1) + E(I)
END DO
```

(a) A Collection of FORTRAN-90 Loops



(b) Loop Communication Graph (LCG)

Figure 1: Graphic Representation of a Collection of Loops

For certain code blocks the loop direction cannot be reversed because of a data dependence across iterations of the loop. Such a dependence is usually referred to as a *loop-carried dependence*. In this case the corresponding node is called a *constrained node*. The node for the loop-carried dependence created by the assignment statement in the last code block in Figure 1 is an example. In an LCG, nodes which are marked with an asterisk refer to constrained nodes, as indicated for Node 6 in Figure (b).

Collective analysis applies to many different loop configurations, each representing a different array-reference sequence; but the methodology we describe considers only two mutually exclusive array-reference sequences at a time. For loops involving vectors (i.e., one-dimensional arrays), the sequence is either ascending or descending loop directions, and for doubly-nested loops that are uniformly ascending, or descending, alternative sequences are determined by interchanging two nested loops within the block, or equivalently, switching the sequence of indices within all access-order vectors of the corresponding block. The way this works is described fully later in the section. For now, however, we merely emphasize that only two array-reference orders, or sequences, are allowed at a time, for each block during each phase of loop compatibility analysis, either *1)* alternative loop directions or *2)* an interchange of two levels of loop nesting. The information about these alternatives is contained in the *Interference Graph* of the loop cluster used during the particular phase of analysis.

An Interference Graph (IG) is an undirected multigraph having nodes grouped according to *blocks*, each block being comprised of two nodes: a *primary node*, n^+, corresponding to a node in the LCG, with input and output ports matching the respective origins and terminuses of incident arcs to this node, and a *compliment node*, n^-, containing a copy of the primary node but with AV elements changed to reflect the alternative array-reference sequence allowed. When loop direction is used as the basis for analysis, the sign of the elements in the AVs are reversed; otherwise, when

2.2 Graphical Constructs

To understand the compatibility analysis process, one first must understand the tools, or constructs, used during the process; the two most important constructs employ graphs. Graphs are often used to depict relationships among individual entities in a way which makes a global relationship clear. A case in point is graph coloring. The classic example is the coloring of, say, political regions on a territorial map. Coloring problems such as map coloring are solved using an *interference graph*. In the case of a map, the regions become nodes and adjacent regional boundaries become arcs. The objective in this case is to color the map interference graph so that no two adjacent nodes are the same color. In general, determining a minimum, or *chromatic number*, is difficult (an NP-complete problem [18]). However, in many situations, coloring heuristics have proven quite practical, as in the case of register allocation [7].

What Sarkar and Gao discovered is that the analysis of loop clusters can also be converted to a graph coloring problem since an analogous correspondence often exists with respect to the orders in which array elements are defined, or generated, within some loops and the order in which they are used, or consumed, in later loops. The objective of coloring in this particular instance is to find a set of compatible loops which produce and consume array elements in a compatible order which allows array values to flow through a cluster of loops in a manner similar to software pipelining.

Collective loop analysis begins with the construction of a *Loop Communication Graph* (LCG). An LCG is a directed multigraph depicting the flow of array values between the code blocks that produce and consume arrays. The "flow" in this case represents flow dependence between assignments to array elements in one code block and the corresponding uses of these elements in follow-on blocks within the loop cluster. Formally, an $LCG = [N, A : A \subseteq N \times N]$, where $n \in N$ is a node and $1 \leq n \leq |N|$. A node n in an LCG represents a code block comprised of perfectly nested loops [28]. An arc $X_{m,n} \in A$ represents the transfer of array elements from a producing code block to a consuming one. Graphically, this transfer is shown by connecting the origin of arc $X_{m,n}$ to the output port of m corresponding to array `X` and the terminus of $X_{m,n}$ to the respective input port of n. An example cluster of loops and corresponding loop communication graph is shown in Figure 1. Nodes in an LCG are further specified by a tuple of Array Access Vectors.

Each *Array Access Vector* (AV) is itself a vector of tuples in which the first element of the tuple indicates the correspondence between the subscripts used to access an array and the loops used to generate these subscripts; the second element indicates the direction of the corresponding loop. A direction element can be either positive (increasing) or negative (decreasing). A positive element is indicated by a plus (+), and a negative element, by a minus (−). An AV contains one loop-direction tuple for each subscript in an array. For example, `A(2+,1−,3+)` describes an array `A` which is referenced within a three-loop nest in which the outer loop causes the second subscript to decrease, the middle loop causes the first subscript to increase, and the inner loop causes the third subscript to increase. An AV $n_x = \{(i, \{'+','-'\})^k : 1 \leq i \leq k\}$ is associated with each input and output port x incident to node n in the LCG, where i is is a loop identifier and k is the depth of loop nesting. When the AV m_x for an array X which is produced in code block m is paired with the AV n_x for an array of the same name which is consumed in another code block n the pair form the specification of the arc $X_{m,n}$ in the LCG. (See Figure 1(b).)

Much work has been accomplished within the past several years with regard to loop optimization, and many effective techniques have emerged [3, 15, 17, 28, 27]. These techniques usually focus on reordering statements either to avoid data and control dependences which can stall a hardware execution pipeline or to avoid long latency memory operations. A key point, however, is that since these techniques usually pertain to a single loop nest, their effect is necessarily local.

In this section a global analysis technique is described to find a *compatible* set of loops across a cluster of loops. By compatible we mean there exists a uniform sequence of array references which allows a cluster of loops to be processed in a software-pipelined manner. A compatible order might be found from ascending and/or descending loop directions or, in certain circumstances, from the interchange of two levels of loop nesting. We refer the the process of finding a compatible set of loops as *collective loop analysis*. In general, collective analysis makes use of several common loop transformations, among which are loop normalization, direction reversal, and loop interchange. If loop analysis is successful, in other words, if a compatible set of loops can be found, several other common transformations to improve the loop cluster become possible, as we will show with a series of brief examples—the primary transformations being loop fusion and array contraction. Other code improving transformations, such as software pipelining, loop unrolling, and instruction scheduling, might also be used to further enhance the quality of the overall code.

2.1 Preconditions

Several conditions are necessary to qualify loops for analysis, and selection of loops satisfying these conditions forms an integral part of the collective analysis process. In selecting loops, the following restrictions apply:

1. The dimensionality of the arrays must be the same for all arrays referenced within the cluster of loops being analyzed.

2. The index variable of all loops must increase or decrease by the same constant amount. Usually this amount, called the loop increment, or *step*, is ± 1.

3. The loops used to define the arrays must be perfectly nested, and the number of loops within each nest must equal the dimensionality of the arrays being referenced. For example, a code block which performs matrix multiplication on conformable square matrices is disallowed because the calculation of the inner product requires an additional loop, one more than the dimensionality of the participating arrays.

4. An index variable can only appear in one subscript position of an array reference. Therefore, references of the type `A(I,I+J)` (in which `I` and `J` are the respective induction variables for a nest of two loops) are disallowed. This condition, that subscripts be *uncoupled*, also ensures that each reference has the same number of subscripts as there are loops in the surrounding loop nest.

5. Although not essential, another assumption we make is that each array element be defined (that is, stored) only once. This condition, called the *single-assignment rule*, is a common characteristic of functional and applicative languages, such as Haskell and SISAL, and it is implicit in array statements used in FORTRAN 90.

1.2 Overview

There are two main parts to this document; the first describes the process of loop-cluster analysis in the case of compatible clusters and the second describes the case of non-compatible clusters. Our discussion of loop analysis begins in Section 2. In this section we first outline the preconditions under which collective loop analysis can be performed. We then describe a variant of Sarkar and Gao's graph-coloring technique which can be used to determine whether the constituent loops within a loop cluster are compatible for collective transformation. A description of the two principle graphical constructs used during this compatibility analysis is given, along with the basic intuition behind the technique. The two principle constructs we describe are *1)* the *Loop Communication Graph* (LCG) and *2)* the loop compatibility *Interference Graph* (IG). Justification for the loop analysis methodology is then provided along with a detailed description. An example of the analysis of a cluster of non-nested loops is then provided, along with a motivating description of several basic follow-on transformations that might be performed to increase loop-cluster performance. The section then continues with an example of another special case, one involving doubly-nested loops, and finally it concludes with a brief description of the strategy for handling clusters of loops which have an arbitrary, but uniform, level of nesting.

In Section 5 we examine the not unlikely case in which compatibility does not exist among the loops of a cluster. In this case, compatible subsets of loops must be identified through a process of cluster partitioning. To motivate our discussion, we first investigate the nature of non-compatibility. Based upon our analysis of some simple situations in which non-compatibility exists, we identify factors which ultimately determine the optimality of cluster partitioning. We then apply our observations to several brief examples which allow us to gain yet a better understanding of the requisite conditions for optimal partitioning. Based upon these examples we derive an efficient heuristic to partition a non-compatible cluster of loops into smaller compatible clusters, using the *flow-augmenting path* algorithm from network flow theory. We then conclude this section with a complete description of our methodology and the rational behind it.

Section 6 describes related research in the area of collective loop transformations. This work is especially important because much of the independent research reported in the previous two sections derives in one way or another from the research reported here. We begin the section with a description of Sarkar and Gao's methodology for collective analysis, describing both its advantages and disadvantages. One important aspect of this particular research is its applicability to multiprocessor situations, a situation we ourselves do not directly address. Next we describe an alternative algorithm for cluster partitioning, called *collective loop fusion*. As part of our description we compare collective loop fusion with the partitioning technique described in Section 5. We then follow this description with a brief survey of related research in the area of loop transformation, and finally we list a number of suggested topics for further research.

2 Analysis of Loop Clusters

Much of the computation involved in scientific programs occurs within loops which produce or consume large arrays. As a result, being able to handle loops efficiently is of fundamental importance.

dataflow architectures. Later, Sarkar and Gao developed strategies for solving several common subclasses of the problem [10, 11, 23]. Most of these subclasses are reviewed throughout this document. In general, the strategy behind Sarkar and Gao's techniques is to use graph coloring to analyze the suitability of a collection of loops to undergo collective transformation. When a cluster of loops can be found which is compatible, the individual loops can be fused, allowing several follow-on transformations to be performed.

The loop transformations employed in identifying and analyzing loop clusters are loop normalization, direction reversal, and loop interchange. *Loop normalization* transforms the loop-control portion of each loop so that all loops in the cluster have uniform bounds, step, and direction. *Loop reversal*, which is often necessary for loop normalization, reverses the order of array element access within a loop body by reversing the indexing function of array references. This transformation is necessary whenever the direction of a loop is changed. *Loop interchange*, on the other hand, changes the order of array referencing by changing the order of nesting of perfectly nested loops. In later sections, illustrations are given of each of these transformations.

Once a compatible loop cluster is identified, loop transformation is performed to improve program performance. Among the follow-on loop transformations which are used to accomplish this objective are loop fusion, array contraction, and software pipelining. *Loop fusion* is accomplished by merging statements from several loops, each having the same loop direction, into a single loop body. The effect is to reduce redundant loop control while increasing loop-body size. With a larger loop body, the need for loop unrolling and the effectiveness of instruction scheduling are also effected. Without fusion, loop unrolling is often used to increase the number of independent instructions within a loop body, available for instruction scheduling. *Loop unrolling* places statements from several loop iterations within a single iteration, increasing the loop step commensurately. *Array contraction* transforms array element references to equivalent scalar references. This transformation can only occur if an array element is both produced and consumed within the same loop body.[1] Contraction has the effect of eliminating array indexing instructions as well as the associated long-latency loads and stores that accompany such references. At the same time, elimination of these array references reduces storage requirements by eliminating the arrays themselves. The impact of array contraction can be substantial, as we will show. *Software pipelining* is an instruction scheduling technique which through an unrolling process finds a pattern of independent instructions across iterations, all of which can be simultaneously executed if the host machine has sufficient parallelism. Not only does this transformation increase the number of instructions which are concurrently executable, it distances long-latency dependencies which otherwise might stall the hardware execution pipeline. As with the preceding transformations, these transformations too are illustrated in subsequent sections.

In cases in which loop compatibility does not exist, the constituent loops of a cluster can usually be partitioned into smaller compatible clusters, and follow-on transformations, such as loop fusion, applied to each smaller cluster. In doing so, much of the original benefit from loop fusion can still be achieved. Loop partitioning for loop fusion is the subject of recent research by Gao, Olsen, Sarkar, and Thekkath, and many of the ideas presented in the document derive from their research, although the specific algorithms we employ are slightly different in several instances [12].

[1] Wolfe uses the term "array contraction" to refer to the technique for reducing the size of a compiler generated temporary array created as a result of scalar expansion [28].

Abstract

Hardware alone is insufficient to allow high-performance computer systems to become successful; sophisticated compilers are also required. Compilers for systems of this type will achieve their effectiveness, not from any single optimizing transformation, but from advanced program analysis and selective application of many proven code-improving transformations. One such analysis technique which holds promise, especially for FORTRAN-90 applications, is a *collective analysis* technique developed by Sarkar and Gao which uses graph coloring to determine whether a cluster of loops can be processed in a pipelined manner [23]. The loop clusters we consider in this case are those consisting of loops which define arrays and successor loops which use them. When array elements which are generated in one loop are able to be referenced in the same order by successor loops, the loops are *compatible* for pipelined processing. On a multiprocessor, pipelined processing is accomplished by running the constituent loops on different processors; whereas on a uniprocessor, the loops are fused and intermediate arrays eliminated through a process of *collective transformation*, itself a process which often creates additional opportunities for code improvement.

In this document we describe enhancements to Sarkar and Gao's loop coloring technique which improve transformation effectiveness of optimizing compilers for uniprocessors; we also describe a minimum-cost partitioning heuristic to achieve efficient transformation in the event a loop cluster contains non-compatible loops.

1 Introduction

1.1 Scope of Analysis

The subject of this memo is a new loop analysis and transformation technique which brings compiler technology closer to meeting the needs of newer high-performance computer architectures. In specific terms, the technique provides a way of analyzing loop clusters in which the lead loops define array elements and successor loops use these elements to define yet other arrays. Array computations of this type could easily comprise a large portion of any scientific computation, especially if the application were written in a language such as FORTRAN 90. Consequently, being able to process arrays of this type efficiently is of fundamental importance. When array elements which are generated in one loop are able to be referenced in the same order by successor loops, the loops are *compatible* for pipelined processing. On a multiprocessor, pipelined processing is accomplished by running the constituent loops on different processors; whereas on a uniprocessor, the loops are fused and intermediate arrays eliminated through a process of *collective transformation*, a process itself which often creates opportunities for additional code improvement.

By considering clusters as a collection, rather than individually, a compiler is able to identify opportunities for transformation which would not be apparent were the loops considered individually, as in traditional loop optimization. As we show in this document, the benefit from applying a global perspective to loop transformation can be substantial, especially in compiling programs, such as FORTRAN-90 programs, which allow statements in vector notation.

The particular loop analysis methods we evaluate are based upon techniques developed by Sarkar and Gao [23]. Gao first raised the issue of collective loop analysis in the context of an *intra-block pipelining* problem, his primary focus being the development of compiler techniques for static

McGill University
School of Computer Science

Collective Analysis and Transformation of Loop Clusters

ACAPS Technical Memo 44
April 24, 1992

Russell Olsen
Guang Gao

Advanced Computer Architecture and Program Structures Group

3480 University St • Montreal • Canada • H3A 2A7