McGill University
School of Computer Science

Optimization of Array Accesses by Collective Loop Transformations

ACAPS Technical Memo 22
December 10, 1990

Guang R. Gao

and

Vivek Sarkar
IBM Palo Alto Scientific Center
1530 Page Mill Road
Palo Alto, CA 94304, USA

Advanced Computer Architecture and Program Structures Group

3480 University St • Montreal • Canada • H3A 2A7

Contents

List of Figures

Abstract

In this paper, we investigate the problem of optimizing array accesses across a collection of loops. We demonstrate that a good solution to such a problem should be based on an optimization scheme, called *collective loop transformations*, that considers all loops simultaneously. In particular, loop reversal, loop interchange and loop fusion are performed collectively on a set of loop nests. The main impact of these transformations is an optimization called *array contraction*, that saves space and time by converting an array variable into a scalar variable or a buffer containing a small number of scalar variables.

This optimization is applicable to general-purpose high-performance architectures. For a multiprocessor architecture, array contraction is performed by executing the producer and consumer loops on separate processors, and by using a smaller buffer for the array communication. For a uniprocessor architecture, array contraction is performed by fusing the producer and consumer loops into a single loop, and using scalar variables for the array communication. In both architectures, our goal is to reconfigure the loops so as to maximize the number of arrays that benefit from array contraction.

The main results of this paper are as follows:

- For one-dimensional loops, we show that the problem of finding the optimal set of loop configurations can be reduced to a two-color graph coloring problem; we present an efficient heuristic algorithm as a solution.
- Once the array contraction decisions have been made, we present an algorithm to allocate minimum extra buffer storage so as to maximize the amount of producer-consumer pipelined parallelism in the loop nests.
- The two-color graph-coloring technique developed for one-dimensional loops is extended to handle two-dimensional loops as well.

For higher (> 2) dimensional loops, the general problem is intractable, but we identify special cases in which a higher dimensional problem can be decomposed into independent one-dimensional or two-dimensional problems. We believe that this kind of problem decomposition is possible for most higher-dimensional loops that are found in practice.

1 Introduction

Loop optimization plays a critical role in compiler optimization of scientific application programs. In such programs, loops are mainly used to define and compute large arrays of values. Processing arrays incurs a large overhead in at least two aspects. First, arrays occupy a large amount of memory space. As an example, a $100 \times 100 \times 100$ array of double-precision floating point numbers requires 8 megabytes of main memory! Second, a large number of load and store operations need to be performed, which becomes a performance bottleneck (also known as the von Neumann bottleneck).

Register allocation has been effectively utilized in code generation to keep frequently used scalar operands in high speed registers. This technique lowers both the latency of operand access, as well as the amount of main memory traffic. Unfortunately, there has not been much success in applying register allocation techniques to subscripted variables. This observation has also been made by other researchers. For example, consider the following paragraph from [CCK90]:

> Although conventional compilation systems do a good job of allocating scalar variables to registers, their handling of subscripted variables leave much to be desired. Most compilers fail to recognize even the simplest opportunities for reuse of subscripted variables.

In this paper, we present an optimization called *array contraction*, that replaces an array variable by a scalar variable or by a buffer containing a small number of scalar variables. Our scheme is based on program analysis of a collection of loops. Each loop produces one or more arrays, using arrays generated by other loops as inputs. When the producer of an array generates the array elements in the same order in which the consumers use them, we have an opportunity to perform array contraction. For a multiprocessor architecture, array contraction is performed by executing the producer and consumer on separate processors, and using a smaller buffer for communicating the array values. For a uniprocessor architecture, array contraction is performed by fusing the producer and consumer loops into a single loop, and using scalar variables for the array communication.

The main results of this paper are:

- For one-dimensional loops, we show that the problem of finding the optimal set of loop configurations can be reduced to a two-color graph coloring problem; we present an efficient heuristic algorithm as a solution.
- Once the array contraction decisions have been made, we present an algorithm to allocate minimum extra buffer storage so as to maximize the amount of producer-consumer pipelined parallelism in the loop nests.
- The two-color graph-coloring technique developed for one-dimensional loops is extended to handle two-dimensional loops as well.

The rest of the paper is organized as follows. Section 2 describes the *loop communication graph*, which is the program representation used by the optimizations. Section 3 formally states the optimization problem. Section 4 provides our solution for the one-dimensional case, based on two-color graph coloring. Section 5 shows how to exploit maximum pipeline parallelism in the loop communication graph, by introducing the minimum amount of extra buffer storage. Section 6 extends the results of Section 4 to the two-dimensional case, and presents a decomposition technique that can often be used to decompose a higher-dimensional problem into independent one-dimensional or two-dimensional problems. Section 7 discusses related work, and Section 8 contains our conclusions.

2 Program Representation

The program representation assumed for performing collective loop transformations is a *Loop Communication Graph* (LCG). A node in the loop communication graph represents a loop nest; an edge represents an array communication between two loop nests. A loop nest is a set of perfectly nested, normalized loops [Wol89].

Array communication is defined by read and write accesses to array variables in the loop body. In this paper, we concentrate on optimizing accesses to functional-style arrays: arrays constructed by loop expressions in applicative languages such as Sisal or Val [MS+85, AD79], or by monolithic array constructors [Hud89, GYDM90], or by Fortran 90 array expressions [For90]. The important property of such arrays is that each array element is written by at most one iteration of the loop nest. This property is also known as the *single-assignment rule* [Hud89]. Arrays whose accesses satisfy the single-assignment rule will be considered as candidates for optimization.

We also make the following assumptions about subscript expressions, for an array to be considered a candidate for optimization:

1. Each subscript expression is of the form, "$\pm$(*index variable*) $\pm$ (*loop-invariant expression*)". This restriction can be easily extended to allow the index variable to have a non-unit coefficient, provided all accesses to the same array dimension have the same coefficient.
2. Each index variable occurs in exactly one dimension of an array access. This implies that the array dimensions are *uncoupled* and that the array has the same number of dimensions as the loop nest. For example, $\mathrm{A}[i_1, i_1 + i_2]$ is a coupled subscript expression, while $\mathrm{A}[i_2, i_1]$ is not. Empirical measurements of real programs have shown that most multi-dimensional array references are uncoupled [ZSY90].

Both assumptions are necessary conditions for array contraction to work. They are satisfied by most Sisal and Val loop expressions, monolithic array constructors, Fortran 90 array expressions, and also by many ordinary Fortran code fragments found in real programs.

Let k be the number of dimensions in an array access that satisfies these conditions. k also equals the number of loops in the enclosing loop nest because of the assumptions made above. The subscript expressions of the array are summarized by an *array access vector* of the form, $({p_1}^{s_1}, \ldots, {p_k}^{s_k})$, where $< p_1, \ldots, p_k >$ is a permutation of $1 \ldots k$, and each s_i is either "+" or "$-$". p_i identifies the index variable used in the i^{th} dimension of the array access; s_i identifies the sign of the index variable in the subscript expression. For example, an array access, $A[i_2, 100 - i_1]$, enclosed within loops `DO` $i_1 = \ldots$ and `DO` $i_2 = \ldots$, will have an array access vector of $< 2^+, 1^- >$.

We do not represent constant terms of the subscript expressions in the array access vector because, in practice, two references with the same array access vector usually have constant terms that are equal or nearly equal. The difference between the constant terms (if any) defines the buffer size necessary for array contraction, but does not impact the choice of loop configurations. We call this kind of buffer an *internal buffer* to distinguish it from the *external buffers* discussed later in Section 5.

An array access vector can have $2^k \times k!$ different configurations. The effect of reversing the i^{th} loop in a loop nest is to toggle the value of s_i in all array access vectors in the loop nest. The effect of permuting the loops of a loop nest (e.g. loop interchange [AK87]) is to permute array access vector entries accordingly.

Array access vectors help identify which arrays can or cannot be optimized. If two read accesses from the same loop nest to the same array have different array access vectors, then the corresponding input array cannot be optimized by array contraction. The loop communication graph defined below represents arrays that are candidates for optimization. Other arrays are assumed to be stored in main memory as usual.

Definition 2.1 A *Loop Communication Graph* (LCG) is an ordered pair of the form (N, E), where

- N is a set of *nodes* that represent *loop nests*. A node (loop nest), $n_a \in N$, has a set of *input ports* numbered $1 \dots I(n_a)$, and a set of *output ports* numbered $1 \dots O(n_a)$, where $I(n_a)$ and $O(n_a)$ are the numbers of the input and output ports respectively.

  These ports identify the input and output arrays of the loop nest represented by the node. Input port i corresponds to an input array variable in the loop, say IA_i. The port is labelled with IA_i's array access vector, which must be the same for all accesses to input array IA_i (otherwise IA_i is not a candidate for array contraction). Similarly, output port i corresponds to an output array variable in the loop, say OA_i. The port is labelled with the OA_i's array access vector; the single-assignment rule guarantees that there is exactly one access to output array OA_i.

- $E \subseteq N \times Z^+ \times N \times Z^+$ is a set of *edges* that represent *array communications* between loop nests. The tuple $(n_a, p_a, n_b, p_b) \in E$ represents a communication from output port p_a of node n_a to input port p_b of node n_b. A communication edge is *consistent* if its input and output ports have the same array access vector; otherwise, it is *inconsistent*. A communication *port* is consistent if all of the communication edges adjacent to it are consistent; otherwise, it is inconsistent. Communication via an inconsistent edge cannot be optimized. Finally, an LCG is consistent if and only if all of its communication edges and ports are consistent. □

Figure 1 shows a Fortran 90 code fragment and its Loop Communication Graph. Loops LA and LB initialize arrays A and B, and therefore have no input edges; all other loops have both input and output edges. Since all loops are one-dimensional in this example, there are only two possible array access vectors, (1^+) and (1^-), represented by the "+" and "−" port labels.

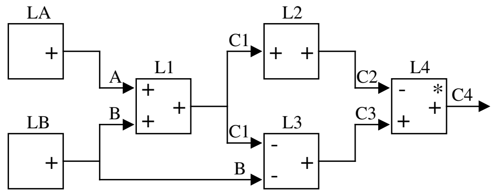
Loop L4 is the only non-reversible loop in Figure 1, because of a loop carried data dependence on array C4. This places a constraint on the set of permissible loop configurations, since loop L4 can only run in the forward direction. Section 4 describes how this constraint is enforced in our optimization algorithm.

Fortran 90 code:

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

C1(1:N:1) = A(1:N:1) + B(1:N:1)
C2(1:N:1) = C1(1:N:1) + 99
C3(1:N:1) = C1(N:1:-1) + B(N:1:-1)
DO I = 1, N
     C4(I) = C4(I-1) + C2(N-I+1) + C3(I)
END DO
```

Loop Communication Graph:



(* = loop L4 must run in the forward direction)

Figure 1: A Fortran 90 code fragment and its LCG

3 Problem Statement

Collective loop transformations for array contraction is performed in two steps:

1. Transform the loop nests so as to minimize the number of *inconsistent* output communication ports in the LCG. This has the effect of maximizing the number of arrays that are optimized by array contraction. In this paper, the transformations considered are *loop reversal*, *loop interchange* and *loop fusion*.
2. Introduce the minimum amount of extra buffer storage in an LCG to exploit maximum pipeline parallelism among loops nest.

The reason for first minimizing the number of inconsistent output communication ports is that each inconsistent output port involves storing an entire array in main memory. As discussed in Section 1, this leads to significant overheads in space (the array storage increases the memory requirements of the application program) and time (the array accesses take longer because the data is stored further away from the processor in the memory hierarchy). After the number of inconsistent output ports has been minimized, we will consider introducing a small (constant) amount of extra storage to balance the program for improved pipelining parallelism.

For the target architecture, we explore two models:

1. A powerful uniprocessor architecture that supports a sufficiently large degree of instruction level parallelism.

2. A multiprocessor with a sufficiently large number of processors and inter-processor communication channels.

Here, "sufficiently" implies that the architecture has enough capacity to exploit all of the parallelism exposed by the collection of loops under consideration. Thus, it appears to the compiler that the machine has unbounded resources. This idealization is similar to other models that have been found to be useful in studying various aspects of compiling techniques [AN90, Gao90]. Even though the absence of resource limits in these models is unrealistic, it is not a serious restriction in our case since we do not allow loop unravelling. Therefore, the amount of available parallelism is limited by the code size of the loop nests and might well be within the reach of realistic architectures. In future work, we will extend our results of to architectures that have more restrictive resource limits.

In our architecture models, pipelining between a pair of adjacent loops in the LCG is an attractive form of parallelism. Since one node can be considered as the producer of an array while the other the consumer of the same array, this form of pipeline parallelism is also known as the *producer-consumer* style of parallelism.

4 Array Contraction for a Collection of One-Dimensional Loops

In this section, we describe our solution to the array contraction problem, assuming that each loop nest has exactly one loop (and each array has exactly one dimension). Extensions for higher dimensions are discussed later in Section 6.

To maximize the number of array contractions performed, we need to minimize the number of inconsistent output ports. For the one-dimensional case, the only loop transformation that can be performed on a loop nest is loop reversal. The set of possible loop configurations can be represented by a *loop configuration vector*, $(s_1, \ldots s_k)$, where s_i = "+" indicates that loop i should be executed in the forward (normalized) direction and s_i = "−" indicates that loop i should be executed in the reverse direction. The initial value of the loop configuration vector is $(+, \ldots, +)$, since all loops are assumed to be normalized. Each array access vector in the loop body can have one of two possible initial values, (1^+) or (1^-), defined with respect to the normalized loop direction. The effect of reversing loop i is to change s_i from "+" to "−" in the loop configuration vector, and to toggle the sign of each array access vector contained within loop i. For a given loop configuration vector, it is easy to recompute all individual array access vectors, and then to count the number of inconsistent output ports, all in linear time. Our goal is to find the loop configuration vector that yields the minimum number of inconsistent output ports. A brute-force solution would take $O(2^{|N|} \times (|N| + |E|))$ time to exhaustively evaluate all possible loop configuration vectors ($|N| = k$ is the number of loop nests in the Loop Configuration Graph). Since this becomes intractable for even moderate values of $|N|$, we present a more efficient approximation algorithm that is provably optimal in interesting special cases.

We show how the problem of minimizing the number of inconsistent output ports can be formulated as a 2-coloring problem for an interference graph that is constructed from the Loop

Communication Graph. If a solution exists to the 2-coloring problem, then it can be used to obtain an optimal loop configuration vector with no inconsistent output ports. Otherwise, we use a heuristic algorithm to find a 2-coloring of a reduced graph, obtained by removing some edges from the interference graph. The goal of this heuristic is similar to the "spill heuristics" employed in graph coloring for register allocation [CAC+81, BCKT89].

We now describe how the Interference Graph is constructed from the Loop Communication Graph. Figure 2 shows the Interference Graph obtained from the Loop Communication Graph in Figure 1. Formally, the Interference Graph $G_I = (N_I, E_I, W_I)$ consists of

- N_I, a set of nodes. For each node in the Loop Communication Graph (say, n_a), we create two nodes in the Interference Graph (called n_a^+ and n_a^-).
- $E_I \subseteq \{\, \{x, y\} | x \in N_I \wedge y \in N_I \,\}$, a set of undirected edges. For each node-pair, n_a^+ and n_a^- in N_I, we add the undirected edge, $\{n_a^+, n_a^-\}$ to E_I. The presence of this edge ensures that n_a^+ and n_a^- will be given different colors. Also, for each communication edge, (n_a, p_a, n_b, p_b), in the Loop Communication Graph, if its input and output ports have the same array access vector then we add an edge between n_a^+ and n_b^-, otherwise we add an edge between n_a^+ and n_b^+ (all interference edges are only added if they do not already exist)[1]. These interference edges ensure that if a 2-coloring is found for IG, then all communication edges in LCG will be made consistent by the loop reversal transformations dictated by the 2-coloring.
- $W_I : E_I \mapsto Z_0^+$, an edge weight mapping from E_I to the set of non-negative integers. If E_I is not 2-colorable, then our goal is to remove a set of interference edges with minimum total weight, so as to make the graph 2-colorable. For the interference edge between n_a^+ and n_a^-, we set $W_I(\{n_a^+, n_a^-\}) = +\infty$, so as to forbid its deletion. For any other interference edge (of the form $\{n_a^+, n_b^-\}$ or $\{n_a^+, n_b^+\}$), we set its weight equal to the number of distinct arrays accessed by different communication edges that correspond to this interference edge. This weight is an estimate of the loss that would be incurred if all the LCG edges corresponding to the interference edge were made inconsistent. We use the number of distinct arrays to provide a simple edge weight function; for greater accuracy, we could sum up the sizes of the distinct arrays.

We first try to obtain a proper coloring of the Interference Graph with 2 colors. Fortunately, 2-coloring is a simpler problem than general k-coloring. There is a simple characterization of all 2-colorable graphs, as shown by the following theorem due to König [Deo74]:

Theorem 4.1 A graph with at least one edge is 2-chromatic if and only if it has no circuits of odd length.

Proof:

[1] Note that edge $\{n_a^-, n_b^+\}$ could be used instead of edge $\{n_a^+, n_b^-\}$ in the Interference Graph, since one implies the other (due to the presence of the $\{n_a^+, n_a^-\}$ and $\{n_b^+, n_b^-\}$ edges). Similarly, edge $\{n_a^-, n_b^-\}$ could be used instead of edge $\{n_a^+, n_b^+\}$. The choice of edges $\{n_a^+, n_b^-\}$ and $\{n_a^+, n_b^+\}$ is arbitrary.

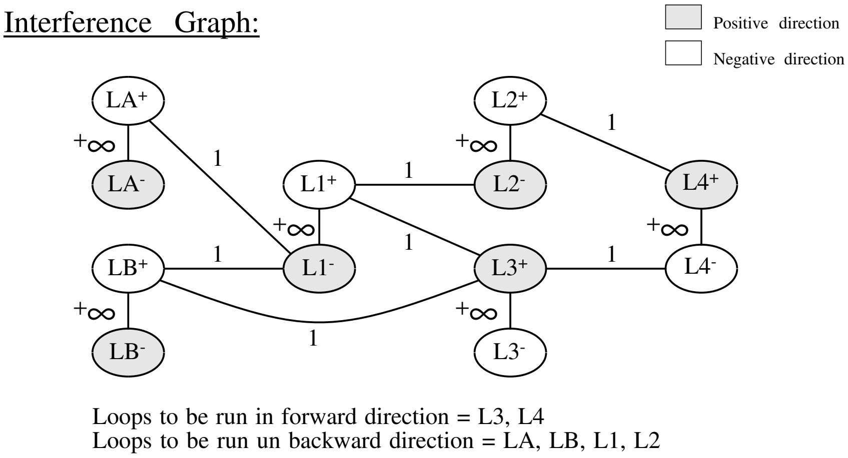


Figure 2: The Interference Graph of the LCG in Figure 1

1. *only if*
   Consider a 2-chromatic graph, G, and assume that G has a circuit, C, of odd length. Then, at least 3 colors are needed for a proper coloring of circuit C. Hence, a proper coloring of graph G will also need at least 3 colors, which implies that G cannot be 2-chromatic. Contradiction.

2. *if*
   Let G be a connected graph with circuits of only even lengths (if the input graph is not connected, we can apply the theorem separately to each of the connected components). Consider a spanning tree T in G, and obtain a proper coloring of T with 2 colors (it is well known how to color a tree with 2 colors — start at any vertex, perform a depth-first search, and assign alternate colors to adjacent vertices). Now, one by one, add the chords of G to T. Since G has no circuits of odd length, the adjacent vertices of each chord must have different colors. Hence the coloring of T also gives a proper 2-coloring of G. □

The characterization in Theorem 4.1 is constructive; it shows how to test (in linear time) if a graph is 2-colorable, and if so, it shows how to obtain a proper 2-coloring (also in linear time). Note that the set of 2-colorable graphs includes all trees, since trees have no circuits at all. We can apply the algorithm described in Theorem 4.1 directly on the Interference Graph, G_I. If G_I is 2-colorable, the color mapping defines an optimal solution with no inconsistent edges in the LCG. Assuming that all loops are reversible, we can arbitrarily pick one color to represent the forward direction (say, color c_1) and another color to represent the reverse direction (say, color c_2). Then, the optimal loop configuration vector is defined as $(s_1, \ldots, s_k)$, where s_i = "+" if node n_i^+ in G_I was colored with c_1, and s_i = "−" if node n_i^+ in G_I was colored with c_2.

This approach can be extended to include loops that are not reversible, by pre-coloring the nodes in the Interference Graph accordingly. For each non-reversible loop (say, loop i) that must run in the forward direction, we pre-assign color c_1 to node n_i^+ in G_I. Similarly, we pre-assign color c_2 to node n_j^+ in G_I, for each non-reversible loop (say, loop j) that must run in the reverse direction. When coloring the spanning tree (as in Theorem 4.1), the only constraint now imposed is that the depth-first search must start at a pre-colored node (if one exists). Then, as usual, we can attempt a 2-coloring and determine if the graph is 2-colorable or not, subject to the constraints imposed by the non-reversible loops.

Consider once again, the LCG shown in Figure 1 and its Interference Graph shown in Figure 2. Since loop L4 must run in the forward direction, we pre-assign color c_1 (shaded box) to node L4$^+$ in Figure 2. Fortunately, the Interference Graph in Figure 2 is 2-colorable. By starting at node L4$^+$ (the pre-colored node), we obtain the coloring shown in Figure 2. Based on the coloring, we determine that loops L3 and L4 must be run in the forward direction, and loops LA, LB, L1, L2 must be run in the backward direction, so as to make the LCG consistent. In this way, arrays A, B, C1, C2, C3 can all be optimized away by array contraction.

If the Interference Graph is not 2-colorable, we have to consider deleting some edges, while minimizing their total weight, so as to make the graph 2-colorable. Our heuristic is to retain the overall structure of the spanning-tree-based coloring algorithm in Theorem 4.1, but to use edge weights to guide the construction of the spanning tree. Specifically, the edge with the largest weight is always the next edge to be included in the spanning tree. Also, to avoid biases due to the choice of the root node when building the spanning tree, the spanning-tree algorithm is performed separately for all possible choices of the root node. The entire algorithm is outlined below:

Algorithm COLOR: 2-coloring of an LCG to minimize the number of inconsistent output ports

Input: A weighted LCG

Output: A loop configuration vector, $(s_1, \ldots, s_k)$

Algorithm:

1. $BESTCOST := +\infty$
2. For each node i in the Interference Graph
   (a) Build a spanning tree, T_i, rooted at node i. Give priority to an edge with the largest weight, when choosing the next edge to include in T_i. This ensures that the $\{n_a^+, n_a^-\}$ edges, with weight $= +\infty$, will always be included in T_i.
   (b) Color T_i with 2 colors, as described in Theorem 4.1 starting at an pre-colored node (if one exists).
   (c) $CURCOST$:= sum of edge weights of all IG edges that connect two nodes with the same color.
   (d) If $CURCOST < BESTCOST$ then set $BESTCOST := CURCOST$ and store the coloring in $COLORMAP$.
3. Use $COLORMAP$ to define the output loop configuration vector, $(s_1, \ldots, s_k)$.

The preceding algorithm has a worst-case $\mathcal{O}(|N_I| \times (|N_I| + |E_I|))$ execution time.

We observe that the 2-coloring algorithm can handle cyclic interference graphs, which means that Algorithm COLOR can also be used for cyclic loop communication graphs. If the LCG is cyclic, then all the feedback edges must be consistent to start with, otherwise the original program would deadlock. To avoid deadlock in the transformed program, we must ensure that the feedback edges remain consistent after our transformations. We enforce this constraint by assigning a cost of $+\infty$ to the interference graph edges that correspond to the LCG's feedback edges.

5 Storage Allocation and Optimization

5.1 External Buffer Storage Between Loops

In this section we consider the second optimization step: allocating the minimum amount of buffer storage to maximize the producer-consumer pipelined parallelism in the loop communication graph. Buffer optimization is performed only on arrays selected for array contraction. Also, if the LCG is cyclic, buffers are only introduced on forward LCG edges, and not on feedback edges.

We illustrate the problem with an example. The LCG shown in Figure 3(a) contains three loops: L1, L2, and L3. Array A is generated by L1 and is consumed by L3 via two different communication paths. On the upper path, A is processed first by loop L2, and the result becomes array B which is sent on to L3. On the lower path, A is routed directly to L3.

Loop L1 (the producer loop of A) could, of course, be allowed to run to completion before loop L2 and L3 (the consumer loops) begin, but this would mean that the entire array of values produced by L1 would need to be stored, defeating the purpose of array contraction. Since our architectural models have sufficient capacity to exploit all of the (static) parallelism in the code, the best choice for the multiprocessor model would be to run all three loops concurrently, in a pipelined manner. As the values of A are generated by L1, they would then be immediately forwarded along the communication paths to L2 and L3, since all array communication edges and ports are consistent.

Since the two paths in Figure 3(a) take different time, temporary storage is needed along the lower communication path to hold some number of values that have been generated by L1 but not yet consumed by L3. The result after optimization is shown in Figure 3(b). In this case, a buffer of size S is added to keep the three loops operating in a maximally pipelined manner, where S is the length of upper path through L2. We call this buffer an *external buffer* since it is between loops, unlike the internal buffers discussed earlier in Section 2.

5.2 An Algorithm for External Buffer Allocation

In this section we present an algorithm to allocate minimum external buffer storage, so as to exploit maximum pipelined parallelism. To discuss the algorithm in graph-theoretic terms, it is convenient to assign weights to edges in the loop communication graph.

Let $G0 = (V, E)$ be an LCG and $G = (V, E, W)$ be its corresponding weighted graph in which each edge $e = (u, p_u, v, p_v)$ in E is weighted by a nonnegative integer weight w_e in W. When a

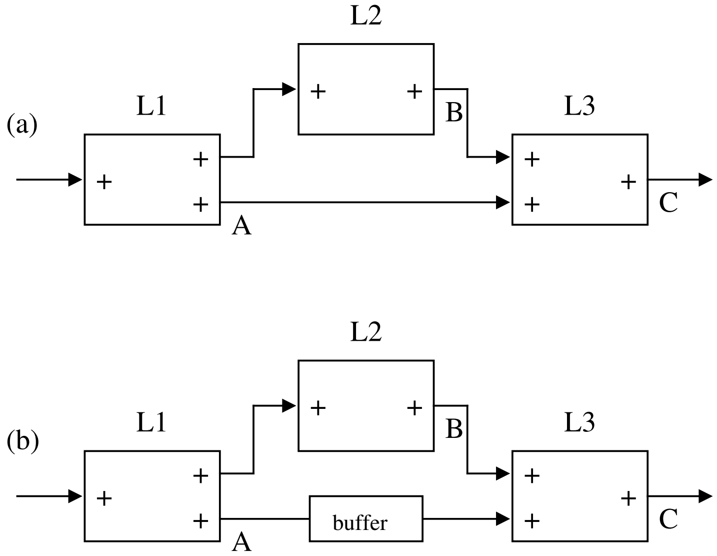


Figure 3: Buffer Allocation for an LCG

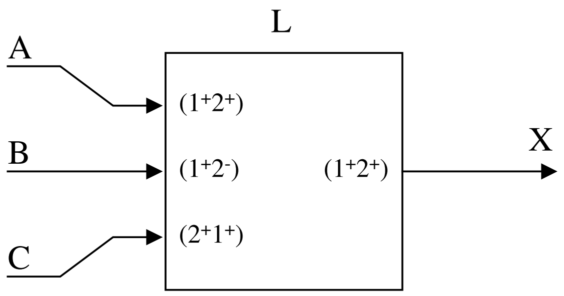
value produced by node u arrives at an input port, p_v, of node v, it is processed via some number of operations before reaching an output port of v—this number is assigned as the weight w_e.

The problem of introducing buffers in a weighted LCG can now be formulated as: Introduce external buffers such that the lengths of any two distinct paths between two nodes are equal; any graph that satisfies this condition is said to be *balanced*. The algorithm must therefore 1) locate the arcs in the graph where buffers should be placed, and 2) determine the buffer size required. Our algorithm is based upon the observation that the length of any longest path from node u to v must remain unchanged whenever a buffer is introduced. Therefore, buffers should be placed on all other paths from u to v to make all paths between u and v have the same length. This property must be enforced for all pairs of nodes.

Without loss of generality, we assume that the weighted LCG has a unique source node, s. An important step in our algorithm is to determine $L(v)$, the length of the longest path from s to v, for each $v \in V$. Once this step is done, a shortcut can be taken to determine the location and size of each buffer to introduce. The key to the shortcut is that the cost of longest path from s to any node v in V is never altered. Let G' be a balanced graph of G as a result of applying some balancing procedure, and let L and L' be the max-cost functions for G and G'. For any node v, $L(v)$=$L'(v)$ should hold after the algorithm is applied. These observations result in a balancing algorithm under which both the locations and sizes of the external buffers can be immediately determined:

Algorithm EBA: External Buffer Allocation for a LCG

Input: A weighted LCG, $G = (V, E, W)$, with a unique source node s.



A, B, C : input arrays
X : output arrays

Loop L :

```
X :=    forall i in [1, n], j in [1, n]
        construct
              A[i-1, j] + B[i, n-j] + C[j, i]
        endall
```

Figure 4: A two dimensional loop and its array access vectors

Output: A balanced graph $G' = (V, E, W')$.

Step 1: Compute the max-cost function L for G.

Step 2: For each edge, $e = (u, p_u, v, p_v)$ in E, introduce an external buffer of size, $L(v) - L(u) - w_e$, on e. Add this size to w_e to obtain $w'_e = L(v) - L(u)$, the new weight in W'.

Step 3: Return the result graph, G'.

Step 1 is equivalent to finding the longest paths from a source node to all other nodes in a directed acyclic graph. This function can be accomplished efficiently using a modified version of Dijkstra's algorithm for finding the shortest paths from a source node to all other nodes [AHU74, Dij59]. The time complexity of the algorithm is $\mathcal{O}(|V||E|)$. For the LCG's found in practice, the degree of each node is usually bounded by a small constant, implying that $E = \mathcal{O}(V)$ and a time complexity for EBA of $\mathcal{O}(|V|^2)$.

6 Extension to Multi-Dimensional Loops

In this section, we examine the problem of minimizing the number of inconsistent output ports in multi-dimensional loops. For the one-dimensional case, the only transformation that could be performed on a loop nest was loop reversal. For multi-dimensional loops, the possible transformations are loop reversal and loop interchange.

Recall that the array access vector of an array reference is represented as $(p_1{}^{s_1}, \ldots, p_k{}^{s_k})$, where $< p_1, \ldots, p_k >$ is a permutation of $1 \ldots k$, and each s_i is either "+" or "−". As an example, Figure 4 shows a two-dimensional loop expression written in the language VAL, along with its corresponding LCG node.

Subsection 6.1 presents a decomposition result which can be used in the multi-dimensional case when all array access vectors have the same permutation. Subsection 6.2 develops a general solution for the two-dimensional case. Finally, subsection 6.3 discusses the problems encountered in solving the general multi-dimensional case.

6.1 Decomposable Multi-Dimensional Loops

Case 1: All array access vectors are of the form $(1^{+/-}, 2^{+/-}, \ldots, k^{+/-})$.

Solving the overall problem one dimension at a time reduces it to k independent one-dimensional problems, and therefore, the solution presented in Section 4 can be applied iteratively on a dimension-by-dimension basis. In general, this type of problem decomposition can be performed when all array access vectors have the same permutation (not necessarily $1 \ldots k$). Although this case is simple, it merits attention because it occurs frequently in real programs.

6.2 Solutions for Two-Dimensional Loops

Case 2: All array access vectors are of the form $(1^+, 2^+)$ or $(2^+, 1^+)$.

In this case, all superscripts have consistent signs. Therefore, the problem can be simplified to interchanging loops so that each pair of ports connected by a communication edge in LCG has the same index permutation (and hence the same array access vector). This permutation problem can also be formulated as a 2-coloring problem, as described below. An example is discussed later in this subsection (Figure 5).

The Interference Graph $G_I = (N_I, E_I, W_I)$ is constructed in a manner similar to Section 4. It consists of

- N_I, a set of nodes. For each node in the LCG (say, n_a), we create two nodes in the Interference Graph (called n_a^+ and n_a^-).
- $E_I \subseteq \{\, \{x, y\} | x \in N_I \wedge y \in N_I \,\}$, a set of undirected edges. For each node-pair, n_a^+ and n_a^- in N_I, we add an edge in E_I to connect them together. For each communication edge, (n_a, p_a, n_b, p_b), in the LCG, if its input and output ports have the same array access vector then we add an edge between n_a^+ and n_b^-, otherwise we add an edge between n_a^+ and n_b^+ (these edges are only added if they do not already exist) Note that "having the same array access vector" just means that their index permutations are consistent. These edges ensure that if a 2-coloring is found for IG, then all communication edges in LCG will be made consistent by the loop permutation transformations dictated by the 2-coloring.

- $W_I : E_I \mapsto Z_0^+$, an edge weight mapping from E_I to the set of non-negative integers, just as in Section 4.

This case is now reduced to the two-color graph coloring problem discussed in Section 4. Colors c_1 and c_2 can represent the (1,2) and (2,1) permutations, respectively. If loop interchange is not permitted for a loop nest (due to dependence constraints), the node for that loop nest should be pre-colored accordingly. Algorithm COLOR from Section 4 is then used as before.

Case 3: All array access vectors are of the form $(1^{+/-}, 2^{+/-})$ or $(2^{+/-}, 1^{+/-})$.

This is the general two-dimensional problem, and is more complicated than Case 2 because the superscripts may be "+" or "−". Our heuristic is to solve the problem in two steps. The first step treats the problem as a Case 2 problem by ignoring the signs of the superscripts. This step attempts to permute the loops in a LCG so that, for each LCG edge, the array access vectors of the source and destination ports attain the same index permutation. All LCG edges that are given consistent source and destination index permutations by the first step, are eligible for optimization by the second step. Observe that the transformed problem is a special case of Case 1 problems with $k = 2$. Therefore, the second step can be be performed using the decomposition technique from subsection 6.1.

To summarize, case 3 provides a general solution for two-dimensional loops. If the LCG has a solution in which no ports are inconsistent, the two-step approach is guaranteed to find the solution; otherwise, the goodness of the solution depends upon the edge-weight values and on how well Algorithm COLOR performs in its choice of interference edges to be deleted (Section 4).

At the top of Figure 5, we see a two-dimensional Fortran code fragment and its LCG. The interference graph for loop interchange is shown next. Since this interference graph is 2-colorable, the coloring defines a new transformed LCG obtained by interchanging the `I` and `J` loops in loop nest L2. Finally, at the bottom of Figure 5, we see the two interference graphs obtained by using Case 1 to decompose the problem of selecting loop directions into one sub-problem per dimension. By coloring these interference graphs, we conclude that loops LA_1, LA_2, LB_2 should be run in the forward direction and loop LB_1 should be run in the backward direction. The loop interchange and the loop reversal together transform the original code so that array A is optimized away by array contraction.

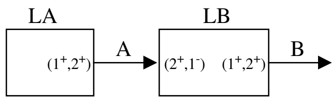
6.3 General Multi-Dimensional Loops

Subsection 6.1 described a special case in which a multi-dimensional problem can be decomposed into separate 1-dimensional problems. In general, when we have a pair of multi-dimensional access vectors for the same array, $({p_1}^{s_1}, \ldots, {p_k}^{s_k})$ and $({q_1}^{t_1}, \ldots, {q_k}^{t_k})$, we can define a *dimension linkage relation*, DLR, which contains ordered pairs of the form (p_i, q_i) and (q_i, p_i). We compute the union of DLR for all pairs of accesses to the same array, and then compute the equivalence classes of the union. Each equivalence class defines an independent sub-problem of the original multi-dimensional problem. The number of sub-problems depends on the degree of linkage among the dimensions.

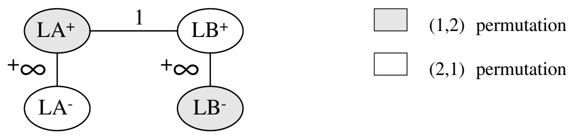
Fortran code:

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

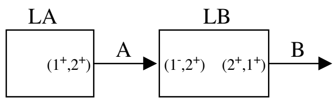
Loop Communication Graph:



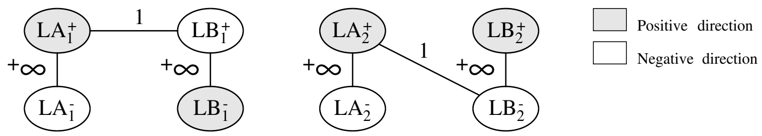
Interference Graph (for loop interchange):



New Loop Communication Graph (after interchange in L2):



Interference Graph (for loop reversal):



Loops to be run in forward direction = LA_1, LA_2, LB_2
Loops to be run un backward direction = LB_1

Figure 5: Array contraction for a two-dimensional LCG

If each sub-problem has at most two dimensions, then we can solve the original multi-dimensional problem by solving the individual sub-problems using the 2-coloring algorithms described earlier.

However, the 2-coloring algorithms do not work for the general multi-dimensional case (with $k > 2$). For example, in a three-dimensional loop, an array access vector has $3! = 6$ possible index permutations. With only two colors, the condition that two nodes have different colors is insufficient to specify what the index permutations should be. Our hope is that, in most cases, the decomposition technique will be able to reduce higher-dimensional problems into independent one-dimensional or two-dimensional sub-problems.

7 Related Work

Most loop transformation techniques described in previous work focus on a single loop nest at a time [KRP$^+$81, AK87, Wol89]. Furthermore, though individual transformations (such as loop reversal and loop interchange) are well understood, the techniques used to combine them are usually ad hoc. Earlier, Kuhn showed how a combination of loop transformations can be used to exploit wavefront parallelism [Kuh80]. More recently, researchers have proposed frameworks for combining loop transformations (mainly: loop reversal, loop interchange, and loop skewing) to achieve optimality for a single loop nest [Ban90, WL90]. Little research has been done so far on optimizing a collection of loops.

Two well known loop transformations that deal with multiple loop nests are loop distribution and loop fusion [Wol89]. Loop fusion is useful for reducing the amount of loop overhead in serial machines. It is also very useful in reducing the memory requirement of loops by reducing the memory-register traffic; this benefit is important for vector machines where vector register loads and stores are expensive. The techniques described in this paper facilitate loop fusion by performing as much array contraction as possible.

The optimization algorithms presented in this paper are novel in their use of two-color graph coloring algorithms. An important feature of our approach is that the optimization is performed on a global scale, by considering several loop nests simultaneously. We believe that this is a powerful approach and that it presents a new set of opportunities for program optimization. The opportunity for optimizing a collection of loops was recognized in [Gao86], but was not explored there.

The partitioning and scheduling work described in [Sar89, SC90, Sar90] collectively optimizes all loops in a procedure, but the loop transformations considered there are loop parallelization, loop chunking and loop fusion. In this paper, we consider loop reversal and loop interchange, which lead to a much larger search space of possible loop configurations. Also, the array contraction optimization described in this paper can be used as both a uniprocessor optimization and a multiprocessor optimization, whereas program partitioning and scheduling deals with the problem of trading off parallelism and overhead in a multiprocessor system.

The problem of minimizing temporary array storage in a single loop nest was discussed as a *sectioning* technique in [All83]. That work also considered minimizing storage for non-functional arrays (i.e. arrays that have storage-related data dependences). However, optimization beyond a single loop nest was not considered.

Finally, the technique of introducing external buffers to exploit pipeline parallelism within a loop body has been studied previously in various different contexts such as systolic structures [Lei83], dataflow architectures [Gao86] and wavefront arrays [KLL86].

8 Conclusions and Future Work

In this paper, we presented a method for performing array contraction by collective loop transformations, and we demonstrated that it is feasible to apply this technique to loops that frequently occur in scientific programs. We expect that the array contraction optimization will help reduce the space and time requirements of scientific programs.

The following is a list of topics for future work:

- Evaluate the effectiveness of array contraction as a practical compiler optimization. Perform experiments on benchmark programs. Compare results and further improve our method, if necessary.
- Consider loop skewing as a possible loop transformation, in addition to loop reversal and loop interchange.
- Extend our work so that it is also applicable to non-functional arrays that may have storage-related data dependences.
- Address the problem of optimizing multi-dimensional arrays, for the cases when decomposition into independent one-dimensional or two-dimensional sub-problems is not possible.
- Evaluate the impact of finite resource limits in the target architecture. If necessary, extend our work to take resource limits into account.

Acknowledgments

We would like to thank Jean-Marc Monti of the Advanced Computer Architecture and Program Structures Group at McGill University for helping with the figures and typesetting the final manuscript. We would also like to acknowledge the support from the National Science and Engineering Research Council (NSERC) and IBM Corporation.

References

[AD79] W. B. Ackerman and J. B. Dennis. VAL—a value-oriented algorithmic language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co., 1974.

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9, 1987.

[All83] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, 1983.

[AN90] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[Ban90] 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. To be published by Pitman in Monographs in Parallel and Distributed Computing.

[BCKT89] Preston Briggs, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):275–284, July 1989.

[CAC+81] 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.

[CCK90] 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.

[Deo74] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall Series in Automatic Computation, 1974.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerishe Mathematik*, 1:269–271, 1959.

[For90] X3J3 Committee of ISO, June 1990. Fortran 90—Draft of the International Standard.

[Gao86] G. R. Gao. A maximally pipelined tridiagonal linear equation solver. *Journal of Parallel and Distributed Computing*, 3(2):215–235, June 1986.

[Gao90] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining*. Kluwer Academic Publishers, Boston, December 1990. To be published.

[GYDM90] G. R. Gao, R. Yates, J. B. Dennis, and L. Mullin. An efficient monolithic array constructor. In *Proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing, Irvine, CA*, 1990. To be published by MIT Press.

[Hud89] P. Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21(3), September 1989.

[KLL86] S. Y. Kung, S. C. Lo, and P. S. Lewis. Timing analysis and optimization of VLSI data flow arrays. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[KRP+81] 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.

[Kuh80] Robert Henry Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980. Report No. UIUCDCS-R-80-1009.

[Lei83] C. E. Leiserson. *Area-Efficient VLSI Computation.* MIT Press, Cambridge, MA, 1983.

[MS+85] J. R. McGraw, S. Skedzielewski, et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 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.

[Sar90] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. Technical report, IBM Research, October 1990. Submitted to special issue of IBM JRD.

[SC90] 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.

[WL90] 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.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* 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 UIUCDCS-R-82-1105, University Illinois at Urbana-Champaign, 1982.

[ZSY90] Zhiyuan Li Zhiyu Shen and Pen-Chung Yew. An emprical study of fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.