Performance Enhancement by Memory Reduction

Yonghong Song
Rong Xu
Cheng Wang
Zhiyuan Li

Purdue University, li@cs.purdue.edu

Report Number:
00-016

Song, Yonghong; Xu, Rong; Wang, Cheng; and Li, Zhiyuan, "Performance Enhancement by Memory Reduction" (2000). Department of Computer Science Technical Reports. Paper 1494.
https://docs.lib.purdue.edu/cstech/1494

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
PERFORMANCE ENHANCEMENT BY MEMORY REDUCTION

Yonghong Song
Rong Xu
Cheng Wang
Zhiyuan Li

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #00-016
November 2000
(Revised January 2001)
Performance Enhancement by Memory Reduction

Yonghong Song  Rong Xu  Cheng Wang  Zhiyuan Li
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{songyh,xur,wangc,li}@cs.purdue.edu

ABSTRACT
In this paper, we propose a technique to reduce the virtual memory required to store program data. Specifically, we present an optimal algorithm to combine loop shifting, loop fusion and array contraction to reduce the temporary array storage required to execute a collection of loops. Memory reduction is formulated as a network flow problem, which is solved by the proposed algorithm in polynomial time. When applied to 20 benchmark programs on two platforms, our technique reduces the memory requirement, counting both the data and the code, by 50% on average. The transformed programs gain a speedup of 1.57 on average, due to the reduced working set and, consequently, the improved data locality. In the best case, a maximum speedup of 41.3 is achieved for one of the benchmark programs.

1. INTRODUCTION
Compiler techniques, such as tiling [29, 30], to exploit temporal data locality within a single loop nest have been studied extensively. However, how to effectively exploit temporal locality between different loop nests remains unclear. This state of the art makes it important to seek locality enhancement techniques beyond tiling.

In this paper, we approach the locality issue by reducing the virtual memory required to store program data. In particular, we seek opportunities to contract the number of dimensions of arrays. For example, a two-dimensional array may be contracted to a single dimension, or a whole array may be contracted to a scalar.

A significant potential benefit, among others, of such a reduction in data size is the increased reuse of the archived data due to the reduced working set.

This paper focuses on reducing the temporary array storage required to execute a collection of loops. The opportunities for such reduction exist often because the most natural way to specify a computation task may not be the most memory-efficient, and because the programs written in array languages such as F90 and HPF are often memory inefficient.

Consider an extremely simple example (Example 1 in Figure 1(a)), where array A is assumed dead after loop L2. After right-shifting loop L2 by one iteration (Figure 1(b)), L1 and L2 can be fused (Figure 1(c)). Array A can then be contracted to two scalars, a1 and a2, as Figure 1(d) shows. (As a positive side-effect, temporal locality of array E is also improved.) The aggressive fusion proposed here also improves temporal data locality between different loop nests.

For a collection of loops defined later in this paper, we formulate the memory reduction problem as a network flow problem, which is optimally solvable in polynomial time.

Additional loop transformations, such as loop interchange and circular loop skewing [30], are used to create opportunities for aggressive fusion.

We have implemented our memory reduction technique in our research compiler. We apply our technique to 20 benchmark programs on two platforms in the experiments. On average, the memory requirement for those benchmarks is reduced by 50%, counting both the code and the data, using the arithmetic mean. The transformed programs have an average speedup of 1.57 (using the geometric mean). A speedup of 41.3 is achieved for one of the benchmarks.

In the rest of this paper, we will present some preliminaries in Section 2. We formulate the network flow problem and prove its complexity in Section 3. We present controlled fusion and discuss enabling techniques in Section 4. Section 5 provides the experimental results, followed by related work and conclusion.

2. PRELIMINARIES
2.1 Program Model
We consider a collection of loop nests, \( L_1, L_2, \ldots, L_m \), \( m \geq 1 \), in their lexical order, as shown in Figure 2(a). The label \( L_i \) denotes a perfect nest of loops with indices \( L_{i,1}, L_{i,2}, \ldots, L_{i,n_i}, n \geq 1 \), starting from the outmost loop. (In Example 1, i.e. Figure 1(a), we have \( m = 2 \) and \( n = 1 \).) Loop \( L_{i,j} \) has the lower bound \( l_{i,j} \) and the upper bound
statements to the first or the last iteration of their adjacent loop nest. The array regions referenced in the given collection of loops are local array regions, respectively, where $L_{i,j}$ and $U_{i,j}$ are loop invariants. For simplicity of presentation, all the loop nests $L_i, 1 \leq i \leq m$, are assumed to have the same nesting level $n$. If they do not have the same nesting level, we can apply our technique to different loop levels incrementally. Figure 3(a) shows a simple example, where the nesting level is 2 for loops $L_1$ and $L_2$ and is 1 for loop $L_3$. We first apply our technique to fuse loops $L_1, I_2$ and $L_3$ at the outmost level only, resulting in the loop nest shown in Figure 3(b). We then apply the technique to loops $L_1$ and $L_2$, resulting in the loop nest in Figure 3(c). If there exist dangling statements between two loop nests, we move them before or after the loop sequence if permitted by the dependences. Otherwise, we perform techniques such as code sinking [30] to move such statements into one of its adjacent loop nest. Alternatively, we can patch these statements to the first or the last iteration of their adjacent loop nest. In the interest of keeping our fundamental idea clear, we do not follow the aforementioned generalizations in this paper. We stay within the model in Figure 2(a) instead.

The array regions referenced in the given collection of loops are divided into three classes:

- **An input array region** is upwardly exposed to the beginning of $L_i$.
- **An output array region** is live after $L_m$.
- **A local array region** does not intersect with any input or output array regions.

By utilizing the existing dependence analysis, region analysis, and live analysis techniques [4, 11, 12, 18], we can compute input, output, and local array regions efficiently. Note that input and output regions can overlap with each other. In Example 1 (Figure 1(a)), $E[0:N]$ is both the input array region and the output array region, and $A[1:N]$ is the local array region. Figure 4(a) shows a more complex example (Example 2), which resembles one of the well-known Livermore loops. In Example 2, where $m = 4$ and $n = 2$, each declared array is of dimension $[N+1,KN+1]$.

Figure 2(b) shows the code form after loop shifting but before loop fusion, where $p'(L_i)$ represents the shifting factor for loop $L_{i,j}$. In the rest of this paper, we assume that loops $L_i$ are coalesced into single level loops [30, 26] after loop shifting but before loop fusion. Figure 2(c) shows the code form after loop coalescing but before loop fusion, and Figure 2(d) shows the code form after loop fusion. The loops are coalesced to ease code generation for general cases. However, in most common cases, loop coalescing is unnecessary [26]. Figure 2(e) shows the code form after loop fusion without loop coalescing applied. Array contraction will then be applied to the code shown in either Figure 2(d) or in Figure 1.

1 Unlike [30], we do not perform loop normalization after coalescing a multi-level loop nest to a single-level one.
2.2 Loop Dependence Graph

We extend the definitions of the traditional dependence distance vector and dependence graph [14] to a collection of loops as follows.

Definition 1. Given a collection of loop nests, \( L_1, \ldots, L_m \), as in Figure 2(a), if a data dependence exists from iteration \((i_1,i_2,\ldots,i_n)\) of loop \( L_1 \) to iteration \((j_1,j_2,\ldots,j_n)\) of loop \( L_2 \), we say the distance vector is \((j_1-i_1,j_2-i_2,\ldots,j_n-i_n)\) for this dependence.

Definition 2. Given a collection of loop nests, \( L_1, L_2, \ldots, L_m \), a loop dependence graph (LDG) is a directed graph \( G = (V,E) \) such that each node in \( V \) represents a loop nest \( L_i \), \( 1 \leq i \leq m \). (We denote \( V = \{ L_1, L_2, \ldots, L_m \} \).) Each directed edge \( e \in E \) represents a data dependence (flow, anti- or output dependence) from \( L_i \) to \( L_j \). The edge \( e \) is annotated by a distance vector \( \mathbf{d}(e) \).

For each dependence edge \( e \), if its distance vector is not constant, we replace it with a set of edges as follows. Let \( \mathbf{d}_0 = (d_1, d_2, \ldots, d_n) \) be the lexicographically minimum distance in \( S \). Let \( S_i = \{ d_i | d_i \leq d, d_i \in S \land (\forall d \in S \exists d \neq d_i) \} \). For any vector \( d_i \in S_i \) (also in \( S \)), there exists no other vector in \( S \) which is no smaller than \( d_i \). We replace the original edge \( e \) with \( (|S_i| + 1) \) edges, annotated by \( d_0 \) and \( d_i \) \((d_i \in S_i, 1 \leq i \leq |S_i|)\) respectively.

Figure 3: Applying to loops not with the same nesting level

Figure 4(b) shows the loop dependence graph for the example in Figure 4(a), without showing the array regions. As an example, the flow dependence from \( L_1 \) to \( L_3 \) with \( \mathbf{d} = (0,0) \) is due to array region \( Z(A(2:JN,2:KN)) \). In Figure 4(b), where multiple dependences of the same type (flow, anti- or output) exist from one node to another, we use one arc to represent them all in the figure. All associated distance vectors are then marked on this single arc.

2.3 Assumptions

We make the following three assumptions in order to simplify our formulation in Section 3.

Assumption 1. The loop trip counts for perfect nests \( L_i \) and \( L_j \) are equal at the same corresponding loop level \( h \), \( 1 \leq h \leq \eta \). This can also be stated as \( u_{i,h} = u_{j,h}, 1 \leq i, j \leq m, 1 \leq h \leq \eta \).

To enforce Assumption 1, one could either partition the iteration spaces of certain loops into equal pieces, or apply loop peeling.

Throughout this paper, we use \( \beta(h) \) to denote the loop trip count of loop \( L_i \) at level \( h \), which is constant or symbolically constant w.r.t. the program segment under consideration. Denote \( \beta = (\beta(1), \ldots, \beta(m)) \). We let \( \sigma^{(n)} = 1 \) and \( \sigma^{(n)} = 1 \leq h \leq n - 1 \). Let \( \sigma = (\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)}) \). In this paper, we also denote \( \tau_n \) as the number of static write references due to local array regions \(^3\) in loop \( L_i \). We arbitrarily assign each static write reference in \( L_i \) a number \( 1 \leq k \leq \tau_n \) in order to distinguish them. Take loops in Figure 4(b) as an example, we have \( \beta = (KN - 1, JN - 1), \sigma = (\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)}) \).

\(^3\)In the rest of this paper, the term of "a static write reference" means "a static write reference due to local array regions".
We make the following assumption about the dependence distance vectors.

**Assumption 2.** The sum of the absolute values of all dependence distances at loop level \( h \) in loop dependence graph \( G = (V,E) \) should be less than one-fourth of the trip count of a loop at level \( h \). This assumption can also be stated as \( \frac{1}{4} |\delta v(e)\| < \frac{1}{2} \beta \) for all \( e \in E \) annotated with the dependence distance vector \( \delta v(e) \).

Assumption 2 is reasonable because for most programs, the constant dependence distances are generally very small. If non-constant dependence distances exist, the techniques discussed in Section 4.2, such as loop interchange and circular loop skewing, may be utilized to reduce such dependence distances.

**Assumption 3.** For each static write reference \( r \), each instance of \( r \) writes to a distinct memory location. No IF-statement guards the statement which contains the reference \( r \). Different static write references write to different portions of main memory.

If a static write reference does not write to a distinct memory location in each loop iteration, we apply scalar or array expansion to this reference \([30]\). Later on, our technique should minimize the total size of the local array regions. In case of IF statements, we assume both branches will be taken. In \([26]\), we discussed the case where the regions written by two different static write references are the same or overlap with each other.

### 2.4 LDG Simplification

The loop dependence graph can be simplified by keeping only dependence edges necessary for memory reduction. The simplification process is based on the following three claims.

**Claim 1.** Any dependence from \( L_i \) to itself is automatically preserved after loop shifting, loop coalescing and loop fusion. This is because we are not reordering the computation within any loop \( L_i \).

**Claim 2.** Among all dependence edges from \( L_i \) to \( L_j \), if \( i \neq j \), suppose that the edge \( e \) has the lexicographically minimum dependence distance vector. After loop shifting and coalescing, if the dependence distance associated with \( e \) is nonnegative, it is legal to fuse loops \( L_i \) and \( L_j \). This is because after loop shifting and coalescing, the dependence distances for all other dependence edges remain equal to or greater than that for the edge \( e \) and thus remain nonnegative. In other words, no fusion-preventing dependences exist. We will prove this claim in Section 3 through Lemma 3.

**Claim 3.** The amount of memory needed to carry a computation is determined by the lexicographically maximum flow-dependence distance vectors which are due to local array regions. We will discuss this claim further in Section 2.5.

During the simplification, we classify all edges into two classes: \( L \)-edges and \( M \)-edges. The \( L \)-edges are used to determine the legality of loop fusion. The \( M \)-edges will determine the minimum memory requirement. All \( M \)-edges are flow dependence edges. But an \( L \)-edge could be a flow, an anti- or an output dependence edge. It is possible that one edge is both an \( L \)-edge and an \( M \)-edge. The simplification process is as follows.

- Based on the claims 1 and 3, for each combination of the node \( L_i \) and the static write reference \( r \) in \( L_i \) where \( r > 0 \), among all dependence edges from \( L_i \) to itself due to \( r \), we keep only the one whose flow dependence distance vector is lexicographically maximum. This edge is an \( M \)-edge.

- Based on the claims 1 and 3, for each node \( L_i \) such that \( r = 0 \), we remove all dependence edges from \( L_i \) to itself.

- Based on the claims 2 and 3, for each node \( L_i \) where \( r > 0 \), among all dependence edges from \( L_i \) to \( L_j \) (\( j \neq i \)), we keep only one dependence edge for legality such that its dependence distance vector is lexicographically minimum. This edge is an \( L \)-edge. For any static write reference \( r \) in \( L_i \), among all dependence edges from \( L_i \) to \( L_j \) (\( j \neq i \)) due to \( r \), we keep only one flow dependence edge whose dependence vector is lexicographically maximum. This edge is an \( M \)-edge.

- Based on the claims 2 and 3, for each node \( L_i \) where \( r = 0 \), among all dependence edges from \( L_i \) to \( L_j \) (\( j \neq i \)), we keep only the dependence edge whose dependence distance vector is lexicographically minimum. This edge is an \( L \)-edge.

The above process simplifies the program formulation and makes graph traversal faster. Figure 4(c) shows the loop dependence graph after simplification of Figure 4(b). In Figure 4(c), we do not mark the classes of the dependence edges. As an example, the dependence edge from \( L_1 \) to \( L_2 \) marked with \((0,0)\) is an \( L \)-edge, and the one marked with \((0,1)\) is an \( M \)-edge. The latter edge is associated with the static write reference \( ZA(J,K) \).

### 2.5 Reference Windows

Loop shifting is applied before loop fusion in order to honor all the dependences. We associate one integer vector \( \hat{p}(L_i) \) with each loop nest \( L_i \) in the loop dependence graph. Denote \( \hat{p}(L_i) = (p^1(L_i), \ldots, p^n(L_i)) \) where \( p^k(L_i) \) is the shifting factor of \( L_i \) at loop level \( k \) (Figure 2(b)). For each dependence edge \(< L_i, L_j > \) with the distance vector \( \delta v \), the new distance vector is \( \hat{p}(L_i) + \delta v - \hat{p}(L_j) \). Our memory minimization problem, therefore, reduces to the problem of determining the shifting factor, \( p^k(L_i) \), for each Loop \( L_i \), such that the total temporary array storage required is minimized after all loops are coalesced and legally fused.

In \([9]\), Gannon et al. use a reference window to quantify the minimum cache footprint required by a dependence with a loop-invariant distance. We shall use the same concept to quantify the minimum temporary storage to satisfy a flow
dependence.

Definition 3. (from [9]) The reference window, \( W(\pi_X) \), for a dependence \( \pi_X : S_1 \rightarrow S_2 \) on a variable \( X \) at time \( t \), is defined as the set of all elements of \( X \) that are referenced by \( S_1 \) at or before \( t \) and will also be referenced (according to the dependency) by \( S_2 \) after \( t \). In Figure 1(a), the reference window due to the flow dependence (from \( L_1 \) to \( L_2 \) due to array \( A \)) in the beginning of each loop \( L_2 \) iteration is \( \{ A(I), A(I + 1), \ldots, A(N) \} \). Its reference window size ranges from 1 to \( N \). In Figure 1(c), the reference window due to the flow dependence (caused by array \( A \)) at the beginning of each loop iteration is \( \{ A(I - 1) \} \). Its reference window size is 1.

Next, we extend Definition 3 for a set of flow dependences as follows.

Definition 4. Given flow dependence edges \( e_1, e_2, \ldots, e_s \), suppose their reference windows at time \( t \) are \( W_1, W_2, \ldots, W_s \) respectively. We define the reference window of \( \{ e_1, e_2, \ldots, e_s \} \) at time \( t \) as \( \cup_{j=1}^s W_j \).

Since the reference window characterizes the minimum memory required to carry a computation, the problem of minimizing the memory required for the given collection of loop nests is equivalent to the problem of choosing loop shifting factors such that the loops can be legally coalesced and fused and that, after fusion, the reference window size of all flow dependences due to local array regions is minimized. Given a collection of loop nests which can be legally fused, we need to predict the reference window after loop coalescing and fusion.

Definition 5. For any loop node \( L_i \) (in an LDG) which writes to local array regions \( R \), suppose iteration \( (j_1, \ldots, j_n) \) becomes iteration \( j \) after loop coalescing and fusion. We define the predicted reference window of \( L_i \) in iteration \( (j_1, \ldots, j_n) \) as the reference window of all flow dependences due to \( R \) in the beginning of iteration \( j \) of the coalesced and fused loop. Suppose the predicted reference window with iteration \( (j_1, \ldots, j_n) \) has the largest size of those due to \( R \). We define it as the predicted reference window size of the entire loop \( L_i \) due to \( R \). We define the predicted reference window due to a static write reference \( r \) in \( L_i \) as the predicted reference window of \( L_i \) due to be the array regions written by \( r \). (For convenience, if \( L_i \) writes to nonlocal regions only, we define its predicted reference window to be empty.)

Based on Definition 5, we have the following lemma:

Lemma 1. The predicted reference window size for the \( k \)th static write reference \( r \) in \( L_1 \) must be no smaller than the predicted reference window size for any flow dependence due to \( r \).

Proof. This is because the predicted reference window size for any flow dependence should be no smaller than the minimum required memory size to carry the computation for that dependence. The predicted reference window size for the \( k \)th static write reference \( r \) in \( L_1 \) should be no smaller than the memory size to carry the computation for all flow dependences due to \( r \).

Theorem 1. Minimizing memory requirement is equivalent to minimizing the predicted reference window size for all flow dependences due to local array regions.

Proof. By Definition 5 and Lemma 1.

In this paper, \( \langle \rangle \) denotes the inner product of \( u \) and \( v \). Given a dependence \( \pi \) with the distance vector \( dv = (d_1, d_2, \ldots, d_n) \) after loop shifting, \( d'v \) is the dependence distance for \( \pi \) after loop coalescing but before loop fusion, which we also call the coalesced dependence distance. Due to Assumption 3, \( d'v \) also represents the predicted reference window size of \( \pi \) both in the coalesced iteration space and in the original iteration space.

Lemma 2. Loop fusion is legal if and only if all coalesced dependence distances are nonnegative.

Proof. This is to preserve all the original dependences.

We now use loop node \( L_2 \) in Figure 4(c) to illustrate how to compute the size of the predicted reference window for one particular static write reference. In this example, the predicted reference window size of \( L_2 \) due to the static write reference \( ZB(J, K) \) is the same as the predicted reference window size of \( L_2 \). There exist two dependence edges from \( L_2 \) to \( L_3 \), one L-edge and one M-edge, with distance vectors \((-1,0) \) and \((0,0) \). There also exist two dependence edges from \( L_2 \) to \( L_4 \), one L-edge and one M-edge. Let

\[
\begin{align*}
\bar{d}v_1 &= \bar{p}(L_3) + (-1,0) - \bar{p}(L_2) \geq \bar{0} \\
\bar{d}v_2 &= \bar{p}(L_4) + (-1,0) - \bar{p}(L_2) \geq \bar{0} \\
\bar{d}v_3 &= \bar{p}(L_4) + (0,0) - \bar{p}(L_2) \geq \bar{0} \\
\bar{d}v_4 &= \bar{p}(L_4) + (0,0) - \bar{p}(L_2) \geq \bar{0}
\end{align*}
\]

Note that loop shifting and coalescing is always legal. To make loop fusion legal, the following constraint is enforced:

\[
\bar{d}d\bar{v}_i \geq 0, 1 \leq i \leq 4
\]

The constraint (5) guarantees that the coalesced dependence distance is nonnegative for all dependences after loop shifting and coalescing but before loop fusion.

\( \bar{d}d\bar{v}_3 \) represents the predicted reference window size for the flow dependence from \( L_2 \) to \( L_3 \), and \( \bar{d}d\bar{v}_4 \) for the predicted
reference window sizes for the flow dependence from \( L_i \) to \( L_j \). The size of the predicted reference window of \( L_j \) can be computed by taking the greater one of the above two reference window sizes, i.e., \( \max(\delta_d v_3, \delta_d v_4) \), according to Lemma 1.

Next, we formulate the objective function for memory reduction to minimize the size of local array regions.

### 3. OBJECTIVE FUNCTION

In this section, we first formulate a graph-based system to minimize the predicted reference window size, thus minimizing the total memory requirement. We then transform our problem to a network flow problem, which is solvable in polynomial time.

Given a loop dependence graph \( G \), the objective function to minimize the size of the predicted reference windows for all loop nests can be formulated as follows. (\( e =< L_i, L_j > \) is an edge in \( G \))

\[
\min(\Sigma_{i=1}^{m} \Sigma_{k=1}^{n} \delta_d M_{i,k})
\]

subject to

\[
\delta_d (p(L_j) + d \delta v(e) - \bar{p}(L_i)) \geq 0, \forall \text{L-edge } e
\]

\[
\delta_d M_{i,k} \geq \delta_d (p(L) + d \delta v(e) - \bar{p}(L)), \forall \text{M-edge } e, 1 \leq k \leq \tau_i
\]

We call the above defined system as Problem 1. In (6), \( \delta_d M_{i,k} \) represents the predicted reference window size for the local array regions due to the \( k \)-th static write reference in \( L_i \).

Constraint (7) says that the coalesced dependence distance must be nonnegative for all L-edges after loop coalescing but before loop fusion. Constraint (8) says that the predicted reference window size, \( \delta_d M_{i,k} \), must be no smaller than the predicted reference window size for every M-edge originated from \( L_i \) and due to the \( k \)-th static write reference in \( L_i \).

Combining the constraint (7) and Assumption 2, the following lemma says that the coalesced dependence distance is also nonnegative for all M-edges.

**Lemma 3.** If the constraint (7) holds, \( \delta_d (\bar{p}(L_j) + d \delta v(e) - \bar{p}(L_i)) \geq 0 \) holds for all M-edges \( e =< L_i, L_j > \) in \( G \).

**Proof.** If \( i = j \), we have \( \delta_d (\bar{p}(L_j) + d \delta v(e) - \bar{p}(L_i)) = \delta_d \delta v(e) \). If \( \delta v(e) = 0 \), then \( \delta_d \delta v(e) = 0 \) holds. Otherwise, assume that the first non-zero component of \( \delta_d \delta v(e) \) is the \( h \)-th component. Based on Assumption 2, we have \( \delta_d \delta v(e) \geq \delta_d (0, 0, 0, 1, -\frac{1}{4} \delta_d (h+1) + 1, \ldots, -\frac{1}{4} \delta_d (n) + 1) > 0 \).

For an M-edge \( e_2 =< L_i, L_j > \), \( i \neq j \), there must exist an L-edge \( e_1 = < L_i, L_j > \). The constraint (7) guarantees that \( \delta_d (\bar{p}(L_j) + d \delta v(e_1) - \bar{p}(L_i)) \geq 0 \) holds. We have \( \delta_d (\bar{p}(L_j) + d \delta v(e_2) - \bar{p}(L_i)) = \delta_d (\bar{p}(L_j) + d \delta v(e_1) - \bar{p}(L_i)) + \delta_d (d \delta v(e_2) - d \delta v(e_1)) \geq \delta_d (d \delta v(e_2) - d \delta v(e_1)) \).

By the definition of L-edges and M-edges, we have \( \delta_d (d \delta v(e_2) - d \delta v(e_1)) \geq \delta_d (d \delta v(e_2) - d \delta v(e_1)) \geq 0 \) holds. Similar to the proof for the case \( i = j \) in the above, we can prove that \( \delta_d (d \delta v(e_2) - d \delta v(e_1)) \geq 0 \) holds. □

From the proof of Lemma 3, we can also see that for any dependence \( \pi \) which is eliminated during our simplification process in Section 2.4, its coalesced dependence distance is also nonnegative, given that the constraint (7) holds. Hence, the coalesced dependence distances for all the original dependences (before simplification in Section 2.4) are nonnegative, after loop shifting and coalescing but before loop fusion. Loop fusion is legal according to Lemma 2.

In Section 2.4, we know that for any flow dependence edge \( e_3 \) from \( L_i \) to \( L_j \) due to the static write reference \( r \) which is eliminated during the simplification process, there must exist an M-edge \( e_4 \) from \( L_i \) to \( L_j \) due to \( r \). From the proof of Lemma 3, \( \delta_d (\bar{p}(L_j) + d \delta v(e_4) - \bar{p}(L_i)) \geq \delta_d (\bar{p}(L_j) + d \delta v(e_2) - \bar{p}(L_i)) \) holds. Hence, the constraint (8) computes the predicted reference window size, \( \delta_d M_{i,k} \), over all flow dependences originated from \( L_i \) due to the \( k \)-th static write reference in the unsimplified loop dependence graph (see Section 2.2). According to Lemma 1, the constraint (8) correctly computes the predicted reference window size, \( \delta_d M_{i,k} \).

#### 3.1 Transforming the Original Problem

We define a new problem, Problem 2, by adding the following two constraints to Problem 1. (\( e =< L_i, L_j > \) is an edge in \( G \))

\[
\delta_d (p(L_j) + d \delta v(e) - \bar{p}(L_i)) \geq 0, \forall \text{L-edge } e
\]

\[
\delta_d M_{i,k} \geq \delta_d (p(L) + d \delta v(e) - \bar{p}(L)), \forall \text{M-edge } e, 1 \leq k \leq \tau_i
\]

In the following, we show that given an optimal solution for Problem 1, we can construct an optimal solution for Problem 2 with the same value for the objective function (6), and vice versa.

**Lemma 4.** Given any optimal solution for Problem 1, we can construct an optimal solution for Problem 2, with the same value for the objective function (6).

**Proof.** The search space of Problem 2 is a subset of that of Problem 1. Given an LDC \( G \), the optimal objective function value (6) for Problem 2 must be equal to or greater than that for Problem 1. Given any optimal solution for Problem 1, we find the shifting factor (\( \bar{p} \)) and \( \delta_d M_{i,k} \) values for Problem 2 as follows.

1. Initially let \( \bar{p} \) and \( \delta_d M_{i,k} \) values from Problem 1 be the solution for Problem 2. In the following steps, we will adjust these values so that all the constraints for Problem 2 are satisfied and the value for the objective function (6) is not changed.

2. If all \( \bar{p} \) values satisfy the constraint (9), go to step 4. Otherwise, go to step 3.
3. This step finds $\vec{p}$ values which satisfy the constraint (9).

Following the topological order of nodes in $G$, find the first node $L_i$ such that there exists an $L$-edge $e = < L_i, L_j >$ where the constraint (9) is not satisfied. (Here we ignore self cycles since they must represent $M$-edges in $G$.) Suppose $\vec{d}^v = \vec{p}(L_j) + \vec{d}^v(e) - \vec{p}(L_i) = (0, \ldots, 0, c_1, \ldots)$ where $c_1 < 0$ is the $a$th and the first nonzero component of $\vec{d}^v$. Let $\vec{d} = (0, \ldots, 0, -c_1, c_1, 0, \ldots)$ where the only two nonzero components are the $a$th and the $(a + 1)$th. Change $\vec{p}(L_i)$ by $\vec{p}(L_i) = \vec{p}(L_i) + \vec{d}$. Because of $\vec{d} = 0$, the new $\vec{p}$ values, including $\vec{p}(L_i)$, satisfy the constraints (7) and (8). The value for the objective function (6) is also not changed.

If $\vec{p}(L_i) + \vec{d}^v(e) - \vec{p}(L_i)$ is still lexicographically negative, we can repeat the above process. Such a process will terminate within at most $n$ times since otherwise the constraint (7) would not hold for the optimal solution of Problem 1.

Note that the node $L_i$ is selected based on the topological order and the shifting factor $\vec{d}(L_i)$ is increased compared to its original value. For any $L$-edge with the destination node $L_j$, if the constraint (9) holds before updating $\vec{p}(L_i)$, it still holds after the update. Such a property will guarantee our process to terminate.

Go to step 2.

4. This step finds $\vec{m}_{i,k}$ values which satisfy the constraint (10).

Given $1 \leq i \leq m$ and $1 \leq k \leq \tau_i$, find the $\vec{m}_{i,k}$ value which satisfies the constraint (10) such that the constraint (10) becomes equal for at least one edge.

If the $\vec{m}_{i,k}$ achieved above satisfies the constraint (8), we are done. Otherwise, we increase the $nth$ component of the $\vec{m}_{i,k}$ value until the constraint (8) holds and becomes equal for at least one edge.

Find all $\vec{m}_{i,k}$ values. The value for the objective function (6) is not changed.

With such $\vec{p}$ and $\vec{m}_{i,k}$ values, the value for the objective function (6) for Problem 2 is the same as that for Problem 1. Hence, we get an optimal solution for Problem 2 with the same value for the objective function (6). $\square$

**Theorem 2.** Any optimal solution for Problem 2 is also an optimal solution for Problem 1.

**Proof.** Given any optimal solution of Problem 2, we take its $\vec{p}$ and $\vec{m}_{i,k}$ values as the solution for Problem 1. Such $\vec{p}$ and $\vec{m}_{i,k}$ values satisfy the constraints (7)-(8), and the value for the objective function (6) for Problem 1 is the same as that for Problem 2. Such a solution must be optimal for Problem 1. Otherwise, we can construct from Problem 1 another solution of Problem 2 which has lower value for the objective function (6), according to Lemma 4. This contradicts to the optimality of the original solution for Problem 2. $\square$

Figure 5: The transformed graph $(G_1)$ for Figure 4(c)

Based on Theorem 2, given an optimal solution for Problem 2, we immediately have an optimal solution for Problem 1. In the rest of this section, we try to solve Problem 2 instead.

By expanding the vectors in Problem 2, an integer programming (IP) problem results. General solutions for IP problems, however, do not take the LDG graphical characteristics into account. Instead of solving the IP problem, we transform it into a network flow problem, as discussed in the rest of this section.

### 3.2 Transforming Problem 2

Given a loop dependence graph $G$, we generate another graph $G_1 = (V_1, E_1)$ as follows.

- For any node $L_i \in G$, create a corresponding node $\bar{L}_i$ in $G_1$.
- For any node $L_i \in G$, if $L_i$ has an outgoing $M$-edge, let the weight of $\bar{L}_i$ be $w(\bar{L}_i) = -\tau_i \vec{d}$. For each static write reference $r_k$ ($1 \leq k \leq \tau_i$) in $L_i$, create another node $\bar{L}_i^{(k)}$ in $G_1$, which is called the sink of $\bar{L}_i$ due to $r_k$. Let the weight of $\bar{L}_i^{(k)}$ be $w(\bar{L}_i^{(k)}) = \vec{d}$.
- For any node $L_i \in G$ which does not have an outgoing $M$-edge, let the weight of $\bar{L}_i$ be $\vec{d}$.
- For any $M$-edge $< L_i, L_j >$ in $G$ due to the static write reference $r_k$, suppose its distance vector $\vec{d}$. Add an edge $< \bar{L}_i, \bar{L}_j^{(k)} >$ to $G_1$ with the distance vector $-\vec{d}$.
- For any L-edge $< L_i, L_j >$ in $G$, suppose its distance vector $\vec{d}$. Add an edge $< \bar{L}_i, \bar{L}_j >$ to $G_1$ with the distance vector $\vec{d}$.

For the original graph in Figure 4(c), Figure 5 shows the transformed graph.

We associate a vector $\vec{d}$ to each node in $G_1$ as follows.

- For each node $\bar{L}_i$ in $G_1$, $\vec{d}_i = \vec{p}(\bar{L}_i)$.
- For each node $\bar{L}_i^{(k)}$, $\vec{d}_i = \vec{m}_{i,k} + \vec{p}(\bar{L}_i)$.

The new system, which we call Problem 3, is defined as follows. ($e = < u, v_j >$ is an edge in $G_1$ annotated by $\vec{d}_k$)}
subject to
\[ q_j - q_i + d_k \geq 0, \forall e \]  
(12)  
\[ \sigma(q_j - q_i + d_k) \geq 0, \forall e \]  
(13)  

**Theorem 3.** Problem 3 is equivalent to Problem 2.

**Proof.** We have
\[ \Sigma_{i=1}^{\vert V \vert} w(v_i)q_i \]  
(11)  
subject to
\[ q_j - q_i + d_k \geq 0, \forall e \]  
(12)  
\[ \sigma(q_j - q_i + d_k) \geq 0, \forall e \]  
(13)  

\[ \Sigma_{k=1}^{\vert E \vert} f(e_k) = w(v_i) + \Sigma_{k=1}^{\vert E \vert} f(e_k), \]  
(17)  
where \( e_k = < v_i, v_j > \) represents an in-edge of \( v_i \) and \( e_k = < v_i, v_j > \) represents an out-edge of \( v_i \).

**Lemma 5.** Given \( G_1 = (V_1, E_1) \), there exists at least one legal network flow.

**Proof.** Find a spanning tree \( T \) of \( G_1 \). Assign the flow value to be 0 for all the edges not in \( T \). Hence, if we can find a legal network flow for \( T \), the same flow assignment is also legal for \( G_1 \).

We assign flow value to the edges in \( T \) in reverse topological order. Since the total weight of the nodes in \( T \) is equal to 0, a legal network flow exists for \( T \).

Based on equation (17), given a legal network flow, we have
\[ \Sigma_{i=1}^{\vert V \vert} w(v_i)q_i = \Sigma_{k=1}^{\vert E \vert} f(e_k)(q_j - q_i) \]  
(18)
where \( e_k = < v_i, v_j > \in E_1 \).

For any node \( v \in V_1 \), we have \( w(v) = \alpha \), where \( \alpha = -\tau, 0 \) or 1. For our network flow algorithm, we abstract out the factor \( \sigma \) from \( w(v) \) such that \( w(v) \) is represented by \( \alpha \) only. Such an abstraction will give each flow value the form \( f(e_k) = c_k \alpha \), where \( c_k \) is an integer constant.

Suppose \( f(e_k) \geq 0 \) for the edge \( e_k \in E_1 \), which is equivalent to \( c_k \geq 0 \). With the constraint (18), we have
\[ f(e_k)(q_j - q_i + d_k) = c_k \sigma(q_j - q_i + d_k) \geq 0. \]  
(19)  

Hence, we have
\[ f(e_k)(q_j - q_i) \geq -f(e_k)d_k. \]  
(20)  

Therefore, with the equation (18), if \( f(e_k) \geq 0 \), we have
\[ \Sigma_{i=1}^{\vert V \vert} w(v_i)q_i \geq -\Sigma_{k=1}^{\vert E \vert} f(e_k)d_k. \]  
(21)  

Collectively, we have the optimality conditions stated as the following theorem such that if they hold, the inequality (21) becomes the equality and the optimality is achieved for Problem 3.

**Theorem 4.** If the following three conditions hold,
1. Constraints (12) and (13) are satisfied, and

2. A legal network flow \( f(e_k) = c_k \sigma \) exists such that \( c_k \geq 0 \) for \( 1 \leq k \leq |E_1| \), and

3. \( \sum_{k=1}^{|E_1|} w(v_i) q_i = -\sum_{k=1}^{|E_1|} f(e_k) d_k \) holds, i.e., inequality (21) becomes an equality.

Problem 3 achieves an optimal solution \(-\sum_{k=1}^{|E_1|} f(e_k) d_k\).

Proof. Obvious from the above discussion. □

3.4 Solving Problem 3

Here, let us consider each vector \( w(u_i), q_i \) and \( d_k \) as a single computation unit. Based on the duality theory [24, 2], Problem 3, excluding the constraint (13), is equivalent to

\[
\begin{align*}
& \text{max} \sum_{i \in E_1} (-f(e_k) d_k) \\
& \text{s.t.} \sum_{i \in E_1} f(e_k) = w(u_i) + \sum_{i \in E_1} f(e_k), 1 \leq i \leq |V_i|, \\
& f(e_i) \geq 0, 1 \leq i \leq |E_1|. 
\end{align*}
\] (22)

The constraint (13) is mandatory for the equivalence between Problem 3 and its dual problem, following the property of duality, Problem 4 achieves an optimal solution only within the search space defined by Problem 3.

Based on the property of duality, Problem 4 achieves an optimal solution if and only if

- The constraints (13), (23) and (24) holds, and
- The objective function values for (11) and (22) are equal, i.e., \( \sum_{i \in E_1} w(u_i) q_i = -\sum_{k=1}^{|E_1|} f(e_k) d_k \) holds.

If we can prove that the constraint (13) holds for the optimal solution of Problem 4, such a solution must also be optimal for Problem 3, according to Theorem 4.

There exist plenty of algorithms to solve Problem 4 [1, 2]. Although these algorithms are targeted to the scalar system (the vector length equals to 1), some of them can be directly adapted to our system by vector summation, subtraction and comparison operations. In [2], the authors present a network simplex algorithm, which can be directly utilized to solve our system. The algorithmic complexity, however, is exponential in the worst case in terms of the number of nodes and edges in \( G_1 \). In [1], the authors present several graph-based polynomial-time algorithms, for example, successive shortest path algorithm with the complexity \( O(|V_i|^3) \), double scaling algorithm with the complexity \( O(|V_i||E_1|log|V_i|) \), and so on. From [1], the current fastest polynomial-time algorithm for solving network flow problem is enhanced capacity scaling algorithm with the complexity \( O((|E_1|log|V_i|)(|E_1| + log|V_i|)) \). For these algorithms, we have the following lemma.

Lemma 6. For any optimal solution of \( q_i \) in Problem 4, there exists a spanning tree \( T \) in \( G_1 \) such that each edge \( e = < u_i, v_j > \) in \( T \) satisfies \( q_i - q_j + d_k \leq 0 \).

Proof. This is true due to the foundation of the simplex method [2]. □

Let \( T \) be the spanning tree in Lemma 6. If we fix any \( q_i \) to be \( 0 \), all \( q_i, 1 \leq i \leq |V_i| \), can be determined uniquely. With such uniquely-determined \( q_i \), we have

\[
|q_i - q_j + d_k| \leq \sum_{l=1}^{|E_1|} |d_k|, 1 \leq i, j \leq |V_i|. 
\] (25)

For any \( e = < u_i, v_j > \in E_1 \) with annotation \( d_k \), with the inequality (25), we have

\[
|q_i - q_j + d_k| \leq |q_i - q_j| + |d_k| \leq \sum_{l=1}^{|E_1|} |d_k|. 
\] (26)

For the inequality (26), based on the inequality (16), we have

\[
|q_i - q_j + d_k| < \beta, e = < u_i, v_j > \in E_1 \text{ is annotated with \( d_k \).} 
\] (27)

Lemma 7. \( \sigma(q_i - q_j + d_k) \geq 0 \), where \( e = < u_i, v_j > \in E_1 \text{ is annotated with } d_k \).

Proof. If \( q_i - q_j + d_k = 0 \), then \( \sigma(q_i - q_j + d_k) \geq 0 \) holds.

Otherwise, assume the first non-zero component is the \( h \)-th for \( q_i - q_j + d_k \). Then, \( q_i^{(h)} - q_j^{(h)} + d_k^{(h)} = 0, 1 \leq h \leq h-1 \), and \( q_j^{(h)} - q_j^{(h)} + d_k^{(h)} > 0 \).

With the constraint (27), we have

\[
\begin{align*}
\sigma(q_i^{(h)} - q_j^{(h)} + d_k^{(h)}) \\
\geq \sigma(0, ..., q_i^{(h)} - q_j^{(h)} + d_k^{(h)}, -q_j^{(h)} + d_k^{(h)} + 1, ..., -\beta^{(h)} + 1) \\
= \sigma^{(h)}(q_i^{(h)} - q_j^{(h)} + d_k^{(h)} - q_j^{(h)} + d_k^{(h)} + \sigma^{(h)} + 1) \\
- \sigma^{(h)}(q_i^{(h)} - q_j^{(h)} + d_k^{(h)} - \sigma^{(h)} + 1) \\
= \sigma^{(h)}(q_i^{(h)} - q_j^{(h)} + d_k^{(h)} - \sigma^{(h)} + 1) + \sigma^{(h)} > 0 
\end{align*}
\]

Hence, Inequality (16) guarantees that the constraint (13) always holds when the optimality of Problem 4 is achieved. The optimal solution for Problem 4 is also an optimal solution for Problem 3.
3.5 Successive Shortest Path Algorithm

We now briefly present one network flow algorithm, successive shortest path algorithm [1], which can be used to solve Problem 4.

The algorithm is depicted in Figure 6. We let \( f(e_k) = f'(e_k) \sigma \) and \( w(u_k) = w'(u_k) \sigma \), where \( f'(e_k) \) and \( w'(u_k) \) are scalars. After the first while iteration, the algorithm always maintains feasible shifting factors and nonnegativity of flow values by satisfying the constraints (12) and (24). It adjusts the flow values such that the constraint (23) holds for all edges in \( G_1 \) when the algorithm ends. For the complete description of the algorithm, including the concept of reduced cost and residue network flow graph, the semantics of sets \( E \) and \( D \), etc., please refer to [1] for details.

We have developed a code generation scheme as well as three linear-time heuristics for fast compilation. Figure 7 shows the transformed code for Example 2 after memory reduction. See [28] for details.

4. REFINEMENTS

4.1 Controlled Fusion

Although array contraction after loop fusion will decrease the overall memory requirement, fusing too many loops can potentially increase the working set size of the loop body, hence it can potentially increase register spilling and cache misses. This is particularly true if a large number of loops are under consideration. To control the number of fused loops, after computing the shifting factors to minimize the memory requirement, we use a simple greedy heuristic, Pick and Reject (see Figure 8), to incrementally select loop nests to be actually fused. If a new addition will cause the estimated cache misses and register spills to be worse than before fusion, then the loop nest under consideration will not be fused. The heuristic then continues to select fusion candidates from the remaining loop nests. The loop nests are examined in an order such that the loops whose fusion saves memory most are considered first. We estimate register spilling by using the approach in [22] and estimate cache misses by using the approach in [7].

It may also be important to avoid fusing too many loop levels if loops are shifted. This is because, after loop shifting, fusing too many loop levels can potentially increase the number of operations due to the IF-statements added in the loop body or due to the effect of loop peeling. Coalescing, if applied, may also introduce more subscript computation overhead. Although all such costs tend to be less significant than the costs of cache misses and register spills, we carefully control the fusion of innermost loops. If the rate of increased operations after fusion exceeds a certain threshold, we only fuse the outer loops.

4.2 Enabling Loop Transformations

We use several well-known loop transformations to enable effective fusion. Long backward data-dependence distances make loop fusion ineffective for memory reduction. Such long distances are sometimes due to incompatible loops [27] which can be corrected by loop interchange. Long backward distances may also be due to circular data dependences which can be corrected by circular loop skewing [27]. Furthermore, our technique applies loop distribution to a node, \( L_i \), if the dependence distance vectors originated from \( L_i \) are different from each other. In this case, distributing the loop may allow different shifting factors for the distributed loops, potentially yielding a more favorable result.

4.3 Tiling vs. Reduction

Suppose the collection of loops in Figure 2(a) are embedded in another loop, \( T \), such that the memory reference foot-
To tile the T loop nest, certain arrays may be duplicated [27]. Let $W_2$ represent the array footprint size of the loop body after the array duplication phase (in the number of data elements) but before tiling. ($W_2$ may then be greater than $W$.) The average cache miss penalty for each T-iteration after tiling will depend on the number of inner-loop levels which are tiled. Based on a detailed calculation [26], we derive the average miss penalty per T-iteration under two-level tiling as

$$
p_1 S_1 W_2 + p_2 S_2 W_2
\frac{C_{b1} B_1}{C_{b2} B_2}
$$

(30)

where $S_1$ and $S_2$ represent the skew factors of tiling and $(B_1, B_2)$ represent the tile size.

The average miss penalty per T-iteration under one-level tiling is estimated as

$$
p_1 W_2 + p_2 S_1 W_2
\frac{C_{b1}}{C_{b2} B_1}
$$

(31)

Which transformation to choose is then determined by a comparison of the estimated cache miss penalties. Our experimental results will show that these simple cost models work quite well.

5. EXPERIMENTAL RESULTS

We have implemented our memory reduction technique in a research compiler, Panorama [12]. We implemented a network flow algorithm, successive shortest path algorithm [1]. The loop dependence graphs in our experiments are relatively simple. The successive shortest path algorithm takes less than 0.06 seconds for each of all the benchmarks. To measure its effectiveness, we tested our memory reduction technique on 20 benchmarks on a SUN Ultra II uniprocessor workstation and on a MIPS R10K processor within an SCI Origin 2000 multiprocessor. The Ultra II processor has a 16K3 directly-mapped L1 data cache with a 16-byte cache line, and it has a 2MB directly-mapped unified L2 cache with a 64-byte cache line. The cache miss penalty is 6 machine cycles for the L1 data cache and 45 machine cycles for the L2 cache. The MIPS R10K has a 32K3 2-way set-associative L1 data cache with a 32-byte cache line, and it has a 4MB 2-way set-associative unified L2 cache with a 128-byte cache line. The cache miss penalty is 9 machine cycles for the L1 data cache and 68 machine cycles for the L2 cache.

5.1 Benchmarks and Memory Reduction

Table 1 lists the benchmarks used in our experiments, their descriptions and their input parameters. In the table, "n/m" represents the number of loops in the loop sequence (m) and the maximum loop nesting level (n). Note that the array size and the iteration counts are chosen arbitrarily for LL14, LL18 and Jacobi. To differentiate two versions of swain in SPEC95 and SPEC2000, we call the SPEC95 version as swain1 and the SPEC2000 version as swain2. swain2 is almost identical to swain1 except for its larger data size. For combustion, we change the array size (N1 and N2) from 1 to 10, so the execution time will last for several seconds. Programs climats, laplace-jb, laplace-ga and all the Purdue set problems are from an HPF benchmark suite at Rice University [20].
Table 1: Test programs

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Description</th>
<th>Input Parameters</th>
<th>m/n</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL14</td>
<td>Livermore Loop No. 14</td>
<td>N = 1001, ITMAX = 60000</td>
<td>3/1</td>
</tr>
<tr>
<td>LL18</td>
<td>Livermore Loop No. 18</td>
<td>N = 400, ITMAX = 100</td>
<td>2/2</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Jacobi Kernel w/o convergence test</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcav</td>
<td>A mesh generation program from SPEC95fp</td>
<td>reference input</td>
<td>5/1</td>
</tr>
<tr>
<td>swim1</td>
<td>A weather prediction program from SPEC95fp</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>swim2</td>
<td>A weather prediction program from SPEC2000fp</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>hydro2d</td>
<td>An astrophysical program from SPEC95fp</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>lucas</td>
<td>A prominence test from SPEC95fp</td>
<td>reference input</td>
<td>10/2</td>
</tr>
<tr>
<td>mg</td>
<td>A multigrid solver from NPB2.3-serial benchmark</td>
<td></td>
<td>2/1</td>
</tr>
<tr>
<td>combustion</td>
<td>A thermochemical program from UM3 Chaos group</td>
<td></td>
<td>1/2</td>
</tr>
<tr>
<td>purdue-02</td>
<td>Purdue set problem02</td>
<td>reference input</td>
<td>2/1</td>
</tr>
<tr>
<td>purdue-03</td>
<td>Purdue set problem03</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>purdue-04</td>
<td>Purdue set problem04</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>purdue-05</td>
<td>Purdue set problem05</td>
<td>reference input</td>
<td>1/2</td>
</tr>
<tr>
<td>purdue-06</td>
<td>Purdue set problem06</td>
<td>reference input</td>
<td>1/2</td>
</tr>
<tr>
<td>purdue-07</td>
<td>Purdue set problem07</td>
<td>reference input</td>
<td>1/2</td>
</tr>
<tr>
<td>purdue-08</td>
<td>Purdue set problem08</td>
<td>reference input</td>
<td>1/2</td>
</tr>
<tr>
<td>purdue-09</td>
<td>Purdue set problem13</td>
<td>reference input</td>
<td>2/2</td>
</tr>
<tr>
<td>climate</td>
<td>A two-layer shallow water climate model from Rice</td>
<td>reference input</td>
<td>2/4</td>
</tr>
<tr>
<td>laplace-jb</td>
<td>Jacobi method of Laplace from Rice</td>
<td>ICYCLE = 500</td>
<td>2/2</td>
</tr>
<tr>
<td>laplace-gs</td>
<td>Gauss-Seidel method of Laplace from Rice</td>
<td>ICYCLE = 500</td>
<td>2/2</td>
</tr>
</tbody>
</table>

Figure 9: Memory sizes before and after transformation on the Ultra II

(Data Size for the Original Programs (unit: KB))

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>96</th>
<th>11520</th>
<th>19360</th>
<th>14750</th>
<th>14794</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LL18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacobi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tomcav</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hydro2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lucas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>combustion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-04</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>purdue-09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>climate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>laplace-jb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>laplace-gs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: Memory sizes before and after transformation on the R10K

21] Except for lucas, all the other benchmarks are written in F77. We manually apply our technique to lucas, which is written in F90. Among 20 benchmark programs, our algorithm finds that all purdue-set programs, lucas, LL14 and combustion do not need to perform loop shifting. For each of the benchmarks in Table 1, all m loops are fused together. For swim1, swim2 and hydro2d, where n = 2, only the outer loops are fused. For all other benchmarks, all n loop levels are fused.

For each of the benchmarks, we examine three versions of the code, i.e. the original one, the one after loop fusion but before array contraction, and the one after array contraction. For all versions of the benchmarks, we use the native Fortran compilers to produce the machine codes. On the Ultra II, we follow the recommendations from SUN's optimizing compiler group and use the following optimization flags. For the original tomcav code, we use "-fast -xchip=ultra2 -xarch=v8plusa -xpad=local:23". For all versions of swim1 and swim2, we use "-fast -xchip=ultra2 -xarch=v8plusa -xpad=common:15". For all versions of combustion, we simply use "-fast" because it produces better-performing codes than using other flags. For all other codes, we use the flag
"fast -xchip=ultra2 -xarch=v8pluse -fsimplc=2". When we compare the best results of different versions, we switch on and off prefetching (i.e. the "-xprefetch" flag) and pick the better result for each version. On the R10K, we simply use the optimization flag "-O3" except with the following adjustments. We switch off prefetching for laplace-jb, software pipelining for laplace-go and loop unrolling for purdue-03. For svml and swm2, the native compiler fails to insert prefetch instructions in the innermost loop body after memory reduction. We manually insert prefetch instructions into the three key innermost loop bodies, following exactly the same prefetching patterns used by the native compiler for the original codes.

Figure 9 compares the code sizes and the data sizes of the original and the transformed codes on the Ultra II. The data size shown for each original program is normalized to 100. The actual data size varies greatly for different benchmarks, which are listed in the table associated with the figure. Similarly, Figure 10 compares the data sizes and the code sizes on the R10K. For mg and climate, the memory requirement differs little before and after the program transformation. This is due to the small size of the contractable local array. For all other benchmarks, our technique reduces the memory requirement noticeably on both machines. The arithmetic mean of the reduction rate, counting both the data and the code, is 50% for all benchmarks on both machines. Specifically, the arithmetic mean is 49% on the Ultra II alone, and 51% on the R10K. For several small purdue benchmarks, the reduction rate is almost 100%.

5.2 Performance

Figure 11 compares the normalized execution time on the Ultra II, where "Mid" represents the execution time of the codes after loop fusion but before array contraction, and "Final" represents the execution time of the codes after array contraction. Similarly, Figure 12 compares the normalized execution time on the R10K. The geometric mean of speedup after memory reduction is 1.57 for all benchmarks on both machines. The geometric mean is 1.73 on the Ultra II alone, and it is 1.40 on the R10K alone.

The best speedup is achieved for program purdue-03, which contains two local arrays, A(1024,1024) and P(1024), which carry values between three adjacent loop nests. Our technique is able to reduce both arrays into scalars and to fuse three loops into one. After comparing the assembly codes on both machines, we found the reason for the less dramatic speedup on the R10K. Prefetching instructions inserted by the native compiler hide memory latency quite well, better than those inserted by the Ultra II's compiler in this case. Excluding program purdue-03 on the Ultra II, the geometric mean of speedup after memory reduction is 1.41 for all other combinations of benchmarks and machines.

We see three programs actually get slowed down slightly after memory reduction. The execution time of both purdue-13 and laplace-gs on the Ultra II is increased by 2%. The execution time of purdue-08 on the R10K is increased by 1%. Both purdue-08 and purdue-13 make several math library function calls which have dominated the execution time. For laplace-gs, loop peeling is applied which may reduce the effectiveness of scalar replacement, and increase the number of total memory references.

Gao and et. al. propose to perform array contraction enabled by loop fusion only [10]. With their technique, the geometric mean of speedup after array contraction is 1.30 for all benchmarks on both machines.

5.3 Memory Reference Statistics

To further understand the effect of memory reduction on the performance, we examined the cache behavior of different versions of the tested benchmarks. We measured the reference count (dynamic load/store instructions), the miss count of the L1 data cache, and the miss count of the L2 unified cache on both machines. We use the perfex package on the MIPS R10K and the perfmon package on the Ultra II to get the cache statistics. Figures 13 and 14 compare such statistics on the Ultra II, where the total reference counts in the original codes are normalized to 100. Similarly, Figures 15 and 16 compare the statistics on the R10K.

When arrays are contracted to scalars, register reuse is often increased. Figures 13 to 16 show that the number of total references get decreased in most of the cases. The total number of reference counts, counting all benchmarks on
Figure 13: Cache statistics before and after transformation on the Ultra II

Figure 14: Cache statistics before and after transformation on the Ultra II (cont.)

Figure 15: Cache statistics before and after transformation on the R10K

Figure 16: Cache statistics before and after transformation on the R10K (cont.)

both machines, is reduced by 21.1% after memory reduction. Specifically, the reduction rate is 20.0% on the Ultra II alone, and it is 22.3% on the R10K alone. However, in a few cases, the total reference counts get increased instead. We examined the assembly codes and found a number of reasons:

1. The fused loop body contains more scalar references in a single iteration than before fusion. This increases the register pressure and sometimes causes more register spilling.

2. The native compilers can perform scalar replacement [3] for references to noncontracted arrays. The fused loop body may prevent such scalar replacement for two reasons:
   - If register pressure is high in a certain loop, the native compiler may choose not to perform scalar replacement.
   - After loop fusion, the array dataflow may become more complex, which then may defeat the native compiler in its attempt to perform scalar replacement.

3. Loop peeling may decrease the effectiveness of scalar replacement since fewer loop iterations benefit from it.

Despite the possibility of increased memory reference counts in a few cases due to the above reasons, Figures 13 to 16 show that cache misses are generally reduced by memory reduction. The total number of cache misses, counting all benchmarks on both machines, is reduced by 58.0% after memory reduction. Specifically, the reduction rate is 28.6% on the Ultra II alone, and it is 63.8% on the R10K alone. The total number of L1 data cache misses, counting all benchmarks on both machines, is reduced by 57.3% after memory reduction. Specifically, the reduction rate is 27.5% on the Ultra II alone, and it is 63.0% on the R10K alone. The improved
cache performance seems to often have a bigger impact on execution time than the total reference counts.

5.4 Interaction with Other Compiler Optimizations

In this subsection, we examine how our memory reduction technique affects prefetching, software pipelining, register allocation and unroll-and-jam. The issue of concern is whether the memory reduction makes other compiler optimizations suffer. A performance comparison with loop tiling is also presented.

Prefetching and Software Pipelining

On the R10K, we compared the performance impact of prefetching and software pipelining on both the original codes and the transformed codes. On the Ultra II, we compared the performance impact of prefetching only, since we cannot specifically switch off software pipelining alone for the native compiler.

Figures 17 and 18 show the normalized execution time with and without prefetching, on the Ultra II, for the original programs and the transformed programs respectively. Prefetching affects the performance little for the transformed codes except tomcatv. Figures 19 and 20 show the normalized
execution time for the original programs, with and without prefetching and software pipelining, on the R10K. Figures 21 and 22 show the normalized execution time for the transformed programs. Software pipelining and prefetching improves the performance for the transformed codes in most cases. One exception is that laplace-jb with prefetching, where prefetching makes performance worse by 49.4%. A close look at cache statistics with perfex shows that prefetching increases the L1 cache miss count by 50% compared with the code without prefetching. Another exception is that laplace-gs with software pipelining, where software pipelining makes performance worse by 20.7%. Based on the results from perfex, software pipelining generates 59% more floating point instructions than without software pipelining (336M vs. 337M).

Register Allocation

As stated earlier in this section, loop fusion may potentially increase register pressure and thus may potentially reduce register reuse. Figures 13 and 14 show that, in 5 of the 20 codes transformed for the Ultra II, slightly more memory references are issued than the original codes. Loop fusion seems to have degraded register reuse somewhat in those codes. However, we should point out that, except in three cases, (laplace-gs on the Ultra II and swim1 and swim2 on the R10K), all those transformed codes in question actually run faster than their original codes. Nonetheless, it is useful to examine the register matter further.

One interesting question is whether the seemingly degraded register utilization is truly due to the increased register pressure. Alternatively, it might be due to the native compiler's inability to properly perform scalar replacement and unroll-and-jam on the fused loop body. (These two techniques are important to good register allocation.) To find the answer, we manually applied unroll-and-jam and scalar replacement to the codes of concern. We experimented with unroll factors from 1 to 4 (a factor of 1 meaning no unrolling), and we applied scalar replacement where possible. We then picked the best results. Figures 23 and 24 show the results on the Ultra II and on the R10K respectively, where "Org" stands for the original code, "Org-Unroll" for the original code with unroll-and-jam plus scalar replacement manually applied. "Trans" stands for the transformed code and "Trans-Unroll" for the transformed code with unroll-and-jam plus scalar replacement applied. From these figures, we conclude that loop fusion indeed increases register pressure somewhat, as unroll-and-jam and scalar replacement applied manually do not seem to make much difference, before or after memory reduction.

Compare with Tiling

As stated in Section 4.3, for certain loop sequences, both tiling and memory reduction may be applied profitably. In our benchmarks, we have LL18, Jacobi, tomsieve, swim1 and swim2 which can be tiled profitably. Table 2 compares the performance between memory reduction and tiling. In this table, Jacobi is tiled at two loop levels. All other four programs are tiled at one loop level only. Even though LL18 can be legally tiled at 2-levels, its performance is poorer than 1-level tiling.

Using the cost estimation in Section 4.3, our research compiler chooses 2-level tiling for Jacobi on both machines. It chooses 1-level tiling for LL18, swim1 and swim2 and chooses
memory reduction for tectav, also on both machines. This turns out to be correct in 9 out of the 10 cases. The exception is L1L8 on the R10K. The tiled assembly code of L1L8 on the R10K shows that the loop index variables of the tile-controlling loop and the time-step loop (i.e. the T loop) are spilled heavily, thus introducing significantly more load/store instructions than the code with memory reduction.

6. RELATED WORK

The work by Fraboulet et al. is the closest to our memory reduction technique [8]. Given a perfectly-nested loop, they use loop alignment to adjust the iteration space for individual statements such that the total buffer size can be minimized. Unlike ours, they only formulate the optimization problem for the 1-D case as a network flow problem, in a form different from ours. For multi-dimensional case, they apply 1-D formulation loop level by loop level. They do not present any experimental results, and they do not consider the effect of memory reduction on cache behavior and execution speed.

Callahan et al. present unroll-and-jam and scalar replacement techniques to replace array references with scalar variables to improve register allocation [3]. However, they only consider the innermost loop in a perfect loop nest. They do not consider loop fusion, neither do they consider array partial contraction. Gao and Sarkar present the collective loop fusion [10]. They perform loop fusion to replace arrays with scalars, but they do not consider partial array contraction. They do not perform loop shifting, therefore they cannot fuse loops with fusion-preventing dependences. Sarkar and Gao perform loop permutation and loop reversal to enable collective loop fusion [23]. These enabling techniques can also be used in our framework.

Lam et al. reduce memory usage for highly-specialized multi-dimensional integral problems where array subscripts are loop index variables [18]. Their program model does not allow fusion-preventing dependences. Lewis et al. proposes to apply loop fusion and array contraction directly in array statement level for those array languages such as F90 [16]. The same result can be achieved if the array statements are transformed into various loops and loop fusion and array contraction are then applied in scalar level. They do not consider loop shifting in their formulation. Strout et al. consider the minimum working set which permits tiling for loops with regular stencil of dependences [28]. Their method applies to perfectly-nested loops only. In [6], Ding indicates the potential of combining loop fusion and array contraction through an example. However, he does not apply loop shift-

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Ultra II</th>
<th>R10K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mem-Red</td>
<td>Tiling</td>
</tr>
<tr>
<td>L1L8</td>
<td>8.3</td>
<td>7.5</td>
</tr>
<tr>
<td>Jacoby</td>
<td>88.2</td>
<td>32.9</td>
</tr>
<tr>
<td>tomcat</td>
<td>80.1</td>
<td>99.0</td>
</tr>
<tr>
<td>avx1</td>
<td>118.4</td>
<td>74.5</td>
</tr>
<tr>
<td>avx2</td>
<td>865</td>
<td>700</td>
</tr>
</tbody>
</table>

7. CONCLUSION

In this paper, we present a locality enhancement technique, memory reduction, which is a combination of loop shifting, loop fusion and array contraction. We reduce the memory reduction problem to a network flow problem, which is solved optimally. (The current fastest algorithm has the complexity $O(|E|log|V|(|E| + |V|)))$ where $G = (V, E)$ is the loop dependence graph.) We propose controlled fusion to prevent excessive register spilling and cache misses which may be caused by excessive loop fusion. We develop a simple memory cost model for memory reduction. For a loop nest where both tiling and memory reduction can apply, the scheme having the smaller cost is chosen. Experimental results so far show that our technique can reduce the memory requirement significantly. At the same time, it speeds up program execution by a factor of 1.57 on average. Furthermore, the memory reduction does not seem to create difficulties for a number of other back-end compiler optimizations. We also believe that memory reduction by itself is vitally important to computers which are severely memory-constrained and to applications which are extremely memory-demanding.

8. REFERENCES


