Data Locality Enhancement by Memory Reduction

Yonghong Song
Sun Microsystems, Inc.
901 San Antonio Rd.
Palo Alto, CA 94303
yonghong.song@eng.sun.com

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

ABSTRACT
In this paper, we propose *memory reduction* as a new approach to data locality enhancement. Under this approach, we use the compiler to reduce the size of the data repeatedly referenced in a collection of nested loops. Between their reuses, the data will more likely remain in higher-speed memory devices, such as the cache. 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. When applied to 20 benchmark programs, our technique reduces the memory requirement, counting both the data and the code, by 51% on average. The transformed programs gain a speedup of 1.40 on average, due to the reduced footprint and, consequently, the improved data locality.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

General Terms
Languages

Keywords
Array contraction, data locality, loop fusion, loop shifting

1. INTRODUCTION
Compiler techniques, such as *tiling* [29, 30], improves temporal data locality by interchanging the nesting order of time-iterative loops and array-sweeping loops. Unfortunately, data dependences in many programs often prevent such loop interchange. Therefore, it is important to seek locality enhancement techniques beyond tiling.

In this paper, we propose *memory reduction* as a new approach to data locality enhancement. Under this approach, we use the compiler to reduce the size of the data repeatedly referenced in a collection of nested loops. Between their reuses, the data will more likely remain in higher-speed memory devices, such as the cache, even without loop interchange. Specifically, we present an optimal algorithm to combine loop shifting, loop fusion and array contraction to contract the number of dimensions of arrays. For examples, a two-dimensional array may be contracted to a single dimension, or a whole array may be contracted to a scalar.

```
L1: DO I = 1, N
      A(I) = E(I) + E(I-1)
    END DO
L2: DO I = 1, N
      E(I) = A(I)
    END DO
```
(a)

```
DO I = 1, N
  A(I) = E(I) + E(I-1)
END DO
DO I = 2, N+1
  E(I-1) = A(I-1)
END DO
```
(b)

```
DO I = 1, N+1
  IF (I.EQ.1) THEN
    A(I) = E(I) + E(I-1)
  ELSE IF (I.EQ.(N+1)) THEN
    E(I-1) = A(I-1)
  ELSE
    A(I) = E(I) + E(I-1)
    E(I-1) = A(I-1)
  END IF
END DO
```
(c)

```
a2 = E(1) + E(0)
DO I = 2, N
  a1 = a2
  a2 = E(I) + E(I-1)
  E(I-1) = a1
END DO
E(N) = a2
```
(d)

Figure 1: Example 1

The opportunities for memory 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.

In [8], Fraboulet *et al.* present a network flow algorithm for memory reduction based on a retiming theory [16]. Given a perfect nest, the retiming technique shifts a number of iterations either to the left or to the right for each statement in the loop body. Different statements may have different shifting factors. Three issues remain unresolved in their work:

- Their algorithm assumes perfectly-nested loops and it applies to one loop level only. Loops in reality, however, are mostly of multiple levels and they can be arbitrarily nested. For perfectly-nested loops, although one may apply their algorithm one level at a time, such an approach does not provably minimize the memory requirement. Program transformations such as *loop coalescing* can coalesce multiple loop levels into a single level. Unfortunately, however, their algorithm is not applicable to the resulting loop, because the required loop model is no longer satisfied.

- Since their work does not target data locality, the relationship between memory minimization and locality/performance enhancement has not been studied. In general, minimization of memory requirement does not guarantee better locality or better performance. Care must be taken to control the scope of loop fusion, lest the increased loop body may increase register spills and cache replacements, even though the footprint of the whole loop nest may be reduced.

- There are no experimental data to show that memory requirement can actually be reduced using their algorithm. Moreover, since their algorithm addresses memory minimization only, there have been no experimental data to verify our conjecture that reduced memory requirement can result in better locality and better performance, especially if the scope of loop fusion is under careful control.

In this paper, we make the following main contributions:

- We present a network flow algorithm which provably minimizes memory requirement for multi-dimensional cases. We handle imperfectly nested loops, which contains a collection of perfect nests, by a combination of loop shifting, loop fusion and array contraction. We completely reformulate the network flow which exactly models the problem for the multi-dimensional general case.

  The work in [8] could also be applied to our program model if loop fusion is applied first, possibly enabled by loop shifting. However, our new algorithm is preferable because (1) for multidimensional cases, our algorithm is optimal and polynomial-time solvable with the same complexity as the heuristic in [8], and (2) our algorithm models imperfectly-nested loops directly.

- For the general case, we propose a heuristic to control fusion such that the estimated numbers of register spills and cache misses should not be greater than the ones in the original loop nests. Even though the benchmarking cases tested so far are too small to utilize such a heuristic, it can be important to bigger cases.

- Many realistic programs may not immediately fit our program model, even though it is already more general than that in [8]. We use a number of compiler algorithms to transform programs in order to fit the model.

- We have implemented our memory reduction technique in our research compiler. We apply our technique to 20 benchmark programs in the experiments. The results not only show that memory requirement can be reduced substantially, but also that such a reduction can indeed lead to better cache locality and faster execution in most of these test cases. On average, the memory requirement for those benchmarks is reduced by 51%, counting both the code and the data, using the arithmetic mean. The transformed programs have an average speedup of 1.40 (using the geometric mean) over the original programs.

In the rest of this paper, we will present 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. Related work is discussed in Section 6 and the conclusion is given in Section 7.

2. PRELIMINARIES

In this section, we introduce some basic concepts and present the basic idea for our algorithm. We present our program model in Section 2.1 and introduce the concept of loop dependence graph in Section 2.2. We make three assumptions for our algorithm in Section 2.3. In Section 2.4, we illustrate the basic idea for our algorithm. In Section 2.5, we introduce the concept of reference window, based on which our algorithm is developed. Lastly in Section 2.6, we show how the original loop dependence graph can be simplified while preserving the correctness of our algorithm.

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 $L_{i,1}$, $L_{i,2}$, $\ldots$, $L_{i,n}$, $n \geq 1$, 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 $u_{i,j}$ 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, we can apply our technique to different loop levels incrementally [27]. Other cases which do not strictly satisfy this requirement are transformed to fit the model using techniques such as code sinking [30].

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_1.
- 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, 19], 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

L_1 : DO $L_{1,1} = l_{1,1}, u_{1,1}$
DO $L_{1,2} = l_{1,2}, u_{1,2}$
...
DO $L_{1,n} = l_{1,n}, u_{1,n}$
...
L_i : DO $L_{i,1} = l_{i,1}, u_{i,1}$
DO $L_{i,2} = l_{i,2}, u_{i,2}$
...
DO $L_{i,n} = l_{i,n}, u_{i,n}$
...
L_m : DO $L_{m,1} = l_{m,1}, u_{m,1}$
DO $L_{m,2} = l_{m,2}, u_{m,2}$
...
DO $L_{m,n} = l_{m,n}, u_{m,n}$

(a)

L_1 : DO $L_{1,1} = l_{1,1} + p^1(L_1), u_{1,1} + p^1(L_1)$
DO $L_{1,2} = l_{1,2} + p^2(L_1), u_{1,2} + p^2(L_1)$
...
DO $L_{1,n} = l_{1,n} + p^n(L_1), u_{1,n} + p^n(L_1)$
...
L_i : DO $L_{i,1} = l_{i,1} + p^1(L_i), u_{i,1} + p^1(L_i)$
DO $L_{i,2} = l_{i,2} + p^2(L_i), u_{i,2} + p^2(L_i)$
...
DO $L_{i,n} = l_{i,n} + p^n(L_i), u_{i,n} + p^n(L_i)$
...
L_m : DO $L_{m,1} = l_{m,1} + p^1(L_m), u_{m,1} + p^1(L_m)$
DO $L_{m,2} = l_{m,2} + p^2(L_m), u_{m,2} + p^2(L_m)$
...
DO $L_{m,n} = l_{m,n} + p^n(L_m), u_{m,n} + p^n(L_m)$

(b)

DO $L_1 = l_1, u_1$
...
DO $L_i = l_i, u_i$
...
DO $L_m = l_m, u_m$
...

(c)

DO $L = min_{i=1}^m l_i, max_{i=1}^m u_i$
...

(d)

DO $J_1 = min_{i=1}^m (l_{i,1} + p^1(L_i)),$
$max_{i=1}^m (u_{i,1} + p^1(L_i))$
DO $J_2 = min_{i=1}^m (l_{i,2} + p^2(L_i)),$
$max_{i=1}^m (u_{i,2} + p^2(L_i))$
...
DO $J_n = min_{i=1}^m (l_{i,n} + p^n(L_i)),$
$max_{i=1}^m (u_{i,n} + p^n(L_i))$

(e)

Figure 2: The original and the transformed loop nests

region and the output array region, and $A[1 : N]$ is the local array region. Figure 3(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 $[JN+1, KN+1]$. *ZP, ZR, ZQ, ZZ, ZA[1,2:KN], ZB[2:JN,KN+1]* are input array regions. *ZP, ZR, ZQ, ZZ* are output array regions. *ZA[2:JN,2:KN]* and *ZB[2:JN,2:KN]* are local array regions.

Figure 2(b) shows the code form after loop shifting but before loop fusion, where $p^j(L_i)$ 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, 27] after loop shifting but before loop fusion. Figure 2(c) shows the code form after loop coalescing but before loop fusion, where l_i and u_i represent the lower and the upper loop bounds for the coalesced loop nest L_i. 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 [27]. 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 2(e).

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.

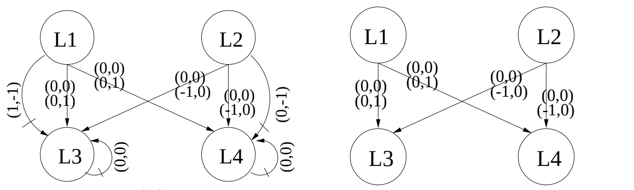
Definition 1. Given a collection of loop nests, $L_1, \ldots, L_m$, as in Figure 2(a), if a data dependence exists from iteration $(i_1, i_2, \ldots, i_n)$ of loop L_1 to iteration $(j_1, j_2, \ldots, j_n)$ of loop L_2, we say the *distance vector* is $(j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$ for this 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, $1 \le i \le m$. (We denote $V = \{L_1, L_2, \ldots, L_m\}$.) Each directed edge, $e = < L_i, L_j >$, 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* [1] $\vec{\mathbf{dv}}(e)$.

For each dependence edge e, if its distance vector is not constant, we replace it with a set of edges as follows. Let S be the set of dependence distances e represents. Let $\vec{\mathbf{d_0}}$ be the lexicographically minimum distance in S. Let $S_1 = \{\vec{\mathbf{d_1}} | \vec{\mathbf{d_1}} \not\succeq \vec{\mathbf{d}}, \vec{\mathbf{d_1}} \in S \wedge (\forall \vec{\mathbf{d}} \in S \wedge \vec{\mathbf{d}} \neq \vec{\mathbf{d_1}})\}$. Any vector

```
L1: DO K = 2, KN
      DO J = 2, JN
         ZA(J, K) = ZP(J − 1, K + 1) + ZR(J − 1, K − 1)
      END DO
    END DO
L2: DO K = 2, KN
      DO J = 2, JN
         ZB(J, K) = ZQ(J − 1, K) + ZZ(J, K)
      END DO
    END DO
L3: DO K = 2, KN
      DO J = 2, JN
         ZP(J, K) = ZP(J, K) + ZA(J, K)
              −ZA(J − 1, K) − ZB(J, K) + ZB(J, K + 1)
      END DO
    END DO
L4: DO K = 2, KN
      DO J = 2, JN
         ZQ(J, K) = ZQ(J, K) + ZA(J, K)
              +ZA(J − 1, K) + ZB(J, K) + ZB(J, K + 1)
      END DO
    END DO
```

(a)



(b) (c)

Figure 3: Example 2 and its original and simplified loop dependence graphs

[1] From [29, 30], $\vec{\mathbf{u}} = (u_1, u_2, \ldots, u_n)$, $\vec{\mathbf{v}} = (v_1, v_2, \ldots, v_n)$, $\vec{\mathbf{u}} + \vec{\mathbf{v}} = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n)$, $\vec{\mathbf{u}} - \vec{\mathbf{v}} = (u_1 - v_1, u_2 - v_2, \ldots, u_n - v_n)$, $\vec{\mathbf{u}} \succ \vec{\mathbf{v}}$ ($\vec{\mathbf{u}}$ is lexicographically greater than $\vec{\mathbf{v}}$) if $\exists 0 \le k \le n-1, (u_1, \ldots, u_k) = (v_1, \ldots, v_k) \wedge u_{k+1} > v_{k+1}$, $\vec{\mathbf{u}} \succeq \vec{\mathbf{v}}$ if $\vec{\mathbf{u}} \succ \vec{\mathbf{v}}$ or $\vec{\mathbf{u}} = \vec{\mathbf{v}}$, $\vec{\mathbf{u}} \ge \vec{\mathbf{v}}$ if $u_k \ge v_k$ $(1 \le k \le n)$.

$\vec{\mathbf{d_1}}$ in S_1 (a subset of S) should lexically be neither smaller than nor equal to any other vector in S. We replace the original edge e with $(|S_1|+1)$ edges, annotated by $\vec{\mathbf{d_0}}$ and $\vec{\mathbf{d_i}}$ ($\vec{\mathbf{d_i}} \in S_1, 1 \le i \le |S_1|$) respectively.

Figure 3(b) shows the loop dependence graph for the example in Figure 3(a), without showing the array regions. As an example, the flow dependence from L_1 to L_3 with $\vec{\mathbf{dv}} = (0,0)$ is due to array region $ZA(2:JN, 2:KN)$. In Figure 3(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, $1 \le h \le n$. This can be also stated as $u_{i,h} - l_{i,h} + 1 = u_{j,h} - l_{j,h} + 1, 1 \le i,j \le m, 1 \le h \le n$.

Assumption 1 can be satisfied by applying techniques such as loop peeling and loop partitioning. If the difference in the iteration counts is small, loop peeling should be quite effective. Otherwise, one can partition the iteration spaces of certain loops into equal pieces.

Throughout this paper, we use $\beta^{(h)}$ to denote the loop trip count of loop L_i at level h, which is constant or symbolicly constant w.r.t. the program segment under consideration. Denote $\vec{\beta} = (\beta^{(1)}, \ldots, \beta^{(n)})$. We let $\sigma^{(n)} = 1$ and $\sigma^{(h)} = \sigma^{(h+1)}\beta^{(h+1)}, 1 \le h \le n-1$. Intuitively, $\sigma^{(h)}$ equals to the number of accumulated level-n loop iterations which will be executed in one level-h loop iteration. Let $\vec{\sigma} = (\sigma^{(1)}, \sigma^{(2)}, \ldots, \sigma^{(n)})$. In this paper, we also denote τ_i as the number of static write references due to local array regions [2] in loop L_i. We arbitrarily assign each static write reference in L_i a number $1 \le k \le \tau_i$ in order to distinguish them. Take loops in Figure 3(b) as an example, we have $\vec{\beta} = (KN-1, JN-1)$, $\vec{\sigma} = (JN-1, 1)$, $\tau_1 = \tau_2 = 1$ and $\tau_3 = \tau_4 = 0$.

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 $\Sigma_{k=1}^{|E|}|\vec{\mathbf{dv}}(e_k)| < \frac{1}{4}\vec{\beta}$ for all $e_k \in E$ annotated with the dependence distance vector $\vec{\mathbf{dv}}(e_k)$.

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.

[2]In the rest of this paper, the term of "a static write reference" means "a static write reference due to local array regions".

(a) (b)

Figure 4: An illustration of memory minimization

Assumption 3. For each static write reference r, each instance of r writes to a distinct memory location. There should be no IF-statement within the loops to guard the statement which contains the reference r. Different static write references should write to different memory locations.

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 [27], we discussed the case in which the regions written by two different static write references are the same or overlap with each other.

2.4 Basic Idea

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

Let $\vec{\mathbf{v}} = \vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}} - \vec{\mathbf{p}}(L_i)$ be the distance vector of $\vec{\mathbf{dv}}$ after loop shifting. In this paper, $\vec{\mathbf{v_1}}\vec{\mathbf{v_2}}$ denotes the inner product of $\vec{\mathbf{v_1}}$ and $\vec{\mathbf{v_2}}$. After loop coalescing, the distance vector $\vec{\mathbf{v}}$ becomes $\vec{\sigma}\vec{\mathbf{v}}$, which we call the *coalesced* dependence distance. In order to make loop fusion legal,

$$\vec{\sigma}\vec{\mathbf{v}} \ge 0 \tag{1}$$

must hold, on which the legality of our transformation stands.

The key to memory minimization is to count the number of simultaneously live local array elements after transformation (loop shifting, loop coalescing and loop fusion). As an example, Figure 4(a) shows part of the iteration space for three loop nests after loop coalescing but before loop fusion. The rectangle bounds the iteration space for each loop nest. Each point in the figure represents one iteration. The two arrows represent the only two flow dependences, d_1 and d_2, which are due to the same static write reference, say r_1. That is, r_1 is the source of d_1 and d_2.

After loop fusion, all the iteration spaces from different loop nests map to a common iteration space. Figure 4(b) shows how three separate iteration spaces map to a common one. Based on Assumption 3, $\vec{\sigma}\vec{\mathbf{v}}$ also represents the number of simultaneously live variables due to $\vec{\mathbf{v}}$ in this common iteration space. In Figure 4(b), the number of simultaneously live variables is 1 for d_1 and is 3 for d_2.

However, there could be overlaps between the simultaneously live local array elements due to different dependences. In Figure 4(b), the simultaneously live array elements for dependences d_1 and d_2 overlap with each other. In this case, the number of simultaneously live local array elements due to the static write reference r_1 will be the greater between the two due to dependences d_1 and d_2, i.e., 3 in this case.

Given a collection of loops, after fusion, the total number of simultaneously live local array elements equals to the sum of the number of simultaneously live local array elements due to each static write reference.

2.5 Reference Windows

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*, $W(\pi_X)_t$ 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 dependence) 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) at the beginning of each loop L2 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 $(\tilde{j_1}, \ldots, \tilde{j_n})$ 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 kth static write reference r in L_i 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 kth static write reference r in L_i 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. □

Given a dependence π with the distance vector $\vec{\mathbf{dv}} = (d^1, d^2, \ldots, d^n)$ after loop shifting, $\vec{\sigma}\vec{\mathbf{dv}}$ is the dependence distance for π after loop coalescing but before loop fusion, which we also call the *coalesced* dependence distance. Due to Assumption 3, $\vec{\sigma}\vec{\mathbf{dv}}$ also represents the predicted reference window size of π 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. □

Take loop node L_2 in Figure 3(c) as an 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, since $ZB(J,K)$ is the only write reference in L2.

2.6 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, $i \neq j$, 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, according to Lemma 1.

During the simplification, we also 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 an edge is classified both as an L-edge and as an M-edge. The simplification process is as follows.

- For each combination of the node L_i and the static write reference r in L_i where $\tau_i > 0$, 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.
- For each node L_i where $\tau_i = 0$, we remove all dependence edges from L_i to itself.
- For each node L_i where $\tau_i > 0$, 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 distance vector is lexicographically maximum. This edge is an M-edge.
- For each node L_i where $\tau_i = 0$, 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 3(c) shows the loop dependence graph after simplification of Figure 3(b). In Figure 3(c), we do not mark the classes of the dependence edges. As an example, the dependence edge from L_1 to L_3 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 $ZA(J, K)$.

3. OBJECTIVE FUNCTION

In this section, we first formulate a graph-based system to minimize the number of simultaneously live local array elements. We then reduce our problem to a network flow problem, which is solvable in polynomial time.

3.1 Formulating Problem 1

Given a loop dependence graph G, the objective function to minimize the number of the simultaneously live local array elements for all loop nests can be formulated as follows. ($e = < L_i, L_j >$ is an edge in G.)

$$min(\Sigma_{i=1}^{m} \Sigma_{k=1}^{\tau_i} \vec{\sigma} \vec{\mathbf{M_{i,k}}}) \tag{2}$$

subject to

$$\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)) \geq 0, \forall \text{ L-edge } e \tag{3}$$

$$\vec{\sigma}\vec{\mathbf{M_{i,k}}} \geq \vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)), \forall \text{ M-edge } e, 1 \leq k \leq \tau_i \tag{4}$$

We call the system defined above as **Problem 1**. In (2), $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$ represents the number of simultaneously live array elements due to the kth static write reference in L_i.

Constraint (3) says that the coalesced dependence distance must be nonnegative for all L-edges after loop coalescing but before loop fusion. Constraint (4) says that the number of simultaneously live local array elements due to the kth static write reference in L_i, $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$, must be no smaller than the number of simultaneously live local array elements for every M-edge originated from L_i and due to the kth static write reference in L_i.

Combining the constraint (3) and Assumption 2, the following lemma says that the coalesced dependence distance is also nonnegative for all M-edges.

LEMMA 3. *If the constraint (3) holds, $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)) \geq 0$ holds for all M-edges $e = < L_i, L_j >$ in G.*

PROOF. If $i = j$, we have $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)) = \vec{\sigma}\vec{\mathbf{dv}}(e)$. If $\vec{\mathbf{dv}}(e) = \vec{\mathbf{0}}$, then $\vec{\sigma}\vec{\mathbf{dv}}(e) = 0$ holds. Otherwise, assume that the first non-zero component of $\vec{\mathbf{dv}}(e)$ is the hth component. Based on Assumption 2, we have $\vec{\sigma}\vec{\mathbf{dv}}(e) \geq \vec{\sigma}(0, \ldots, 0, 1, -\frac{1}{4}\beta^{(h+1)} + 1, \ldots, -\frac{1}{4}\beta^{(n)} + 1) > 0$.

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 (3) guarantees that $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_1) - \vec{\mathbf{p}}(L_i)) \geq 0$ holds. We have $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_2) - \vec{\mathbf{p}}(L_i)) = \vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_1) - \vec{\mathbf{p}}(L_i)) + \vec{\sigma}(\vec{\mathbf{dv}}(e_2) - \vec{\mathbf{dv}}(e_1)) \geq \vec{\sigma}(\vec{\mathbf{dv}}(e_2) - \vec{\mathbf{dv}}(e_1))$.

By the definition of L-edges and M-edges, we have $\vec{\mathbf{dv}}(e_2) - \vec{\mathbf{dv}}(e_1) \succeq \vec{\mathbf{0}}$. Similar to the proof for the case of $i = j$ in the above, we can prove that $\vec{\sigma}(\vec{\mathbf{dv}}(e_2) - \vec{\mathbf{dv}}(e_1)) \geq 0$ holds. □

From the proof of Lemma 3, we can also see that for any dependence π which is eliminated during our simplification process in Section 2.6, its coalesced dependence distance is also nonnegative, given that the constraint (3) holds. Hence, the coalesced dependence distances for all the original dependences (before simplification in Section 2.6) are nonnegative, after loop shifting and coalescing but before loop fusion. Loop fusion is legal according to Lemma 2.

In Section 2.6, 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, $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_4) - \vec{\mathbf{p}}(L_i)) \geq \vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_3) - \vec{\mathbf{p}}(L_i))$ holds. Hence, the constraint (4) computes the predicted reference window size, $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$, over all flow dependences originated from L_i due to the kth static write reference in the unsimplified loop dependence graph (see Section 2.2). According to Lemma 1, the constraint (4) correctly computes the predicted reference window size, $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$.

3.2 Transforming Problem 1

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.)

$$\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i) \succeq \vec{\mathbf{0}}, \forall \text{ L-edge } e \qquad (5)$$

$$\vec{\mathbf{M_{i,k}}} \succeq \vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i), \forall \text{ M-edge } e, 1 \leq k \leq \tau_i \quad (6)$$

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 (2).*

PROOF. The search space of **Problem 2** is a subset of that of **Problem 1**. Given an LDG G, the optimal objective function value (2) 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 ($\vec{\mathbf{p}}$) and $\vec{\mathbf{M_{i,k}}}$ values for **Problem 2** as follows.

1. Initially let $\vec{\mathbf{p}}$ and $\vec{\mathbf{M_{i,k}}}$ 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 (2) is not changed.

2. If all $\vec{\mathbf{p}}$ values satisfy the constraint (5), go to step 4. Otherwise, go to step 3.

3. This step finds $\vec{\mathbf{p}}$ values which satisfy the constraint (5).

   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 (5) is not satisfied. (Here we ignore self cycles since they must represent M-edges in G.) Suppose $\vec{\mathbf{dv}}' = \vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i) = (0, \ldots, 0, c_1, \ldots)$ where $c_1 < 0$ is the sth and the first nonzero component of $\vec{\mathbf{dv}}'$. Let $\vec{\delta} = (0, \ldots, 0, -c_1, c_1\beta^{(s+1)}, 0, \ldots)$ where the only two nonzero components are the sth and the $(s+1)$th. Change $\vec{\mathbf{p}}(L_j)$ by $\vec{\mathbf{p}}(L_j) = \vec{\mathbf{p}}(L_j) + \vec{\delta}$. Because of $\vec{\sigma}\vec{\delta} = 0$, the new $\vec{\mathbf{p}}$ values, including $\vec{\mathbf{p}}(L_j)$, satisfy the constraints (3) and (4). The value for the objective function (2) is also not changed.

   If $\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)$ is still lexicographically negative, we can repeat the above process. Such a process will terminate within at most n times since otherwise the constraint (3) 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 $\vec{\mathbf{p}}(L_j)$ is increased compared to its original value. For any L-edge with the destination node L_j, if the constraint (5) holds before updating $\vec{\mathbf{p}}(L_j)$, it still holds after the update. Such a property will guarantee our process to terminate.

   Go to step 2.

4. This step finds $\vec{\mathbf{M_{i,k}}}$ values which satisfy the constraint (6).

   Given $1 \leq i \leq m$ and $1 \leq k \leq \tau_i$, find the $\vec{\mathbf{M_{i,k}}}$ value which satisfies the constraint (6) such that the constraint (6) becomes equal for at least one edge.

   If the $\vec{\mathbf{M_{i,k}}}$ achieved above satisfies the constraint (4), we are done. Otherwise, we increase the nth component of the $\vec{\mathbf{M_{i,k}}}$ value until the constraint (4) holds and becomes equal for at least one edge.

   Find all $\vec{\mathbf{M_{i,k}}}$ values. The value for the objective function (2) is not changed.

With such $\vec{\mathbf{p}}$ and $\vec{\mathbf{M_{i,k}}}$ values, the value for the objective function (2) 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 (2). □

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 $\vec{\mathbf{p}}$ and $\vec{\mathbf{M_{i,k}}}$ values as the solution for **Problem 1**. Such $\vec{\mathbf{p}}$ and $\vec{\mathbf{M_{i,k}}}$ values satisfy the constraints (3)-(4), and the value for the objective function (2) 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 (2), according to Lemma 4. This contradicts to the optimality of the original solution for **Problem 2**. □

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,

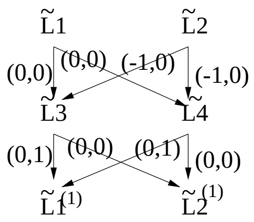


Figure 5: The transformed graph (G_1) for Figure 3(c)

we transform it into a network flow problem, as discussed in the next subsection.

3.3 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 $\tilde{L_i}$ in G_1.
- For any node $L_i \in G$, if L_i has an outgoing M-edge, let the weight of $\tilde{L_i}$ be $w(\tilde{L_i}) = -\tau_i \vec{\sigma}$. For each static write reference r_k $(1 \le k \le \tau_i)$ in L_i, create another node $\tilde{L_i}^{(k)}$ in G_1, which is called the *sink* of $\tilde{L_i}$ due to r_k. Let the weight of $\tilde{L_i}^{(k)}$ be $w(\tilde{L_i}^{(k)}) = \vec{\sigma}$.
- For any node $L_i \in G$ which does not have an outgoing M-edge, let the weight of $\tilde{L_i}$ be $\vec{\mathbf{0}}$.
- For any M-edge $< L_i, L_j >$ in G due to the static write reference r_k, suppose its distance vector is $\vec{\mathbf{dv}}$. Add an edge $< \tilde{L_j}, \tilde{L_i}^{(k)} >$ to G_1 with the distance vector $-\vec{\mathbf{dv}}$.
- For any L-edge $< L_i, L_j >$ in G, suppose its distance vector is $\vec{\mathbf{dv}}$. Add an edge $< \tilde{L_i}, \tilde{L_j} >$ to G_1 with the distance vector $\vec{\mathbf{dv}}$.

For the original graph in Figure 3(c), Figure 5 shows the transformed graph.

We assign a vector $\vec{\mathbf{q}}$ to each node in G_1 as follows.

- For each node $\tilde{L_i}$ in G_1, $\vec{\mathbf{q_i}} = \vec{\mathbf{p}}(L_i)$.
- For each node $\tilde{L_i}^{(k)}$, $\vec{\mathbf{q_i}} = \vec{\mathbf{M_{i,k}}} + \vec{\mathbf{p}}(L_i)$.

The new system, which we call **Problem 3**, is defined as follows. ($e =< v_i, v_j >$ is an edge in G_1 annotated by $\vec{\mathbf{d_k}}$.)

$$min \Sigma_{i=1}^{|V_1|} w(v_i) \vec{\mathbf{q_i}} \tag{7}$$

subject to

$$\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}} \succeq \vec{\mathbf{0}}, \forall e \tag{8}$$

$$\vec{\sigma}(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) \ge 0, \forall e \tag{9}$$

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

PROOF. We have

$\Sigma_{i=1}^{|V_1|} w(v_i) \vec{\mathbf{q_i}}$

$= \Sigma_{\tau_i > 0}(-\tau_i \vec{\sigma} \vec{\mathbf{p}}(L_i) + \Sigma_{k=1}^{\tau_i}(\vec{\sigma}(\vec{\mathbf{M_{i,k}}} + \vec{\mathbf{p}}(L_i)))) + \Sigma_{\tau_i = 0} \vec{\mathbf{0}} \vec{\sigma}$

$= \Sigma_{i=1}^{m} \Sigma_{k=1}^{\tau_i} (\vec{\mathbf{M_i}} \vec{\sigma})$.

Hence the objective function (2) is equivalent to (7).

For each edge $e =< \tilde{L_i}, \tilde{L_j} >$ in G_1, the inequality (8) is equivalent to

$$\vec{\mathbf{p}}(L_j) - \vec{\mathbf{p}}(L_i) + \vec{\mathbf{dv}}(e_1) \succeq \vec{\mathbf{0}}, \tag{10}$$

where e_1 is an L-edge in G from L_i to L_j. Inequality (10) is equivalent to (5), hence inequality (8) is equivalent to (5).

For each edge $e =< \tilde{L_j}, \tilde{L_i}^{(k)} >$ in G_1, the inequality (8) is equivalent to

$$\vec{\mathbf{M_{i,k}}} + \vec{\mathbf{p}}(L_i) - \vec{\mathbf{p}}(L_j) - \vec{\mathbf{dv}}(e_1) \succeq \vec{\mathbf{0}}, \tag{11}$$

where e_1 is an M-edge in G from L_i to L_j due to the kth static write reference in L_i. Inequality (11) is equivalent to (6), hence inequality (8) is equivalent to (6).

Similarly, it is easy to show that the constraints (3) and (4) are equivalent to constraint (9). □

Note that one edge in G could be both an L-edge and an M-edge, which corresponds to two edges in G_1. From Assumption 2, the following inequality holds for the transformed graph G_1:

$$\Sigma_{k=1}^{|E_1|} |\vec{\mathbf{dv}}(e_k)| < \frac{1}{2} \vec{\beta}, \tag{12}$$

where $e_k \in E_1$ is annotated with the dependence distance vector $\vec{\mathbf{dv}}(e_k)$.

Same as **Problem 2**, **Problem 3** can be solved by linearizing the vector representation so that the original problem becomes an integer programming problem, which in its general form, is NP-complete. In the next subsections, however, we show that we can achieve an optimal solution in polynomial time for **Problem 3** by utilizing the network flow property.

3.4 Optimality Conditions

We develop optimality conditions to solve **Problem 3**. We utilize the network flow property. A network flow consists of a set of vectors such that each vector $f(e_i)$ corresponds to an edge $e_i \in E_1$ and, for each node $v_i \in V_1$, the sum of flow values from all the in-edges should be equal to $w(v_i)$ plus the sum of flow values from all the out-edges. That is,

$$\Sigma_{e_k =< ., v_i > \in E_1} f(e_k) = w(v_i) + \Sigma_{e_k =< v_i, . > \in E_1} f(e_k), \tag{13}$$

where $e_k =< ., v_i >$ represents an in-edge of v_i and $e_k =< v_i, . >$ 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 $\vec{0}$ 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 $\vec{0}$, a legal network flow exists for T. $\square$

Based on equation (13), given a legal network flow, we have

$$\Sigma_{i=1}^{|V_1|} w(v_i)\vec{\mathbf{q_i}} = \Sigma_{k=1}^{|E_1|} f(e_k)(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}}) \tag{14}$$

where $e_k =< v_i, v_j >\in E_1$.

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

Suppose $f(e_k) \succeq \vec{0}$ for the edge $e_k \in E_1$, which is equivalent to $c_k \geq 0$. With the constraint (9), we have

$$f(e_k)(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) = c_k\vec{\sigma}(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) \geq 0. \tag{15}$$

Hence, we have

$$f(e_k)(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}}) \geq -f(e_k)\vec{\mathbf{d_k}}. \tag{16}$$

Therefore, with the equation (14), if $f(e_k) \succeq \vec{0}$, we have

$$\Sigma_{i=1}^{|V_1|} w(v_i)\vec{\mathbf{q_i}} \geq -\Sigma_{k=1}^{|E_1|} f(e_k)\vec{\mathbf{d_k}}. \tag{17}$$

Collectively, we have the optimality conditions stated as the following theorem such that if they hold, the inequality (17) becomes the equality and the optimality is achieved for **Problem 3**.

THEOREM 4. *If the following three conditions hold,*

1. *Constraints (8) and (9) are satisfied, and*
2. *A legal network flow $f(e_k) = c_k\vec{\sigma}$ exists such that $c_k \geq 0$ for $1 \leq k \leq |E_1|$, and*
3. *$\Sigma_{i=1}^{|V_1|} w(v_i)\vec{\mathbf{q_i}} = -\Sigma_{k=1}^{|E_1|} f(e_k)\vec{\mathbf{d_k}}$ holds, i.e., inequality (17) becomes an equality.*

then **Problem 3** *achieves an optimal solution* $-\Sigma_{k=1}^{|E_1|} f(e_k)\vec{\mathbf{d_k}}$.

PROOF. Obvious from the above discussion. $\square$

3.5 Solving Problem 3

Here, let us consider each vector $w(v_i)$, $\vec{\mathbf{q_i}}$ and $\vec{\mathbf{d_k}}$ as a single computation unit. Based on the duality theory [24, 2], **Problem 3**, excluding the constraint (9), is equivalent to

$$max_{k=1}^{|E_1|}(-f(e_k)\vec{\mathbf{d_k}}) \tag{18}$$

subject to

$$\Sigma_{e_k=<.,v_i>\in E_1} f(e_k) =$$

$$w(v_i) + \Sigma_{e_k=<v_i,.>\in E_1} f(e_k), 1 \leq i \leq |V_1|. \tag{19}$$

$$f(e_i) \succeq \vec{0}, 1 \leq i \leq |E_1|. \tag{20}$$

The constraint (9) is mandatory for the equivalence between **Problem 3** and its dual problem, following the development of optimality conditions in Section 3.4 [1]. The constraint (19) in the dual system precisely defines a flow property, where each edge e_i is associated with a flow vector $f(e_i)$. We define **Problem 4** as the system by (7)-(8) and (18)-(20). Similar to $w(v_i)$, the vector $f(e_k)$ is represented by c_k where $f(e_k) = c_k\vec{\sigma}$. Although apparently the search space of **Problem 4** encloses that of **Problem 3**, **Problem 4** has correct solutions 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 (8), (19) and (20) holds, and
- The objective function values for (7) and (18) are equal, i.e., $\Sigma_{i=1}^{|V_1|} w(v_i)\vec{\mathbf{q_i}} = -\Sigma_{k=1}^{|E_1|} f(e_k)\vec{\mathbf{d_k}}$ holds.

If we can prove that the constraint (9) 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 those 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. A network simplex algorithm [2] 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. Several graph-based algorithms [1], on the other hand, have polynomial-time complexity. Examples include *successive shortest path algorithm* with the complexity $O(|V_1|^3)$, *double scaling algorithm* with the complexity $O(|V_1||E_1|log|V_1|)$, and so on. From [1], the current fastest polynomial-time algorithm for solving network flow problem is *enhanced capacity scaling algorithm* with the complexity $O((|E_1|log|V_1|)(|E_1| + log|V_1|)$. For these algorithms, we have the following lemma.

LEMMA 6. *For any optimal solution of* $\vec{\mathbf{q_i}}$ *in* **Problem 4**, *there exists a spanning tree* T *in* G_1 *such that each edge* $e =< v_i, v_j >$ *in* T *satisfies* $\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}} = \vec{\mathbf{0}}$.

PROOF. This is true due to the foundation of the simplex method [2]. $\square$

Let T be the spanning tree in Lemma 6. If we fix any $\vec{\mathbf{q}}$ to be $\vec{\mathbf{0}}$, all $\vec{\mathbf{q_i}}, 1 \leq i \leq |V_1|$, can be determined uniquely. With such uniquely-determined $\vec{\mathbf{q_i}}$, we have

$$|\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}}| \leq \Sigma_{s=1}^{|E_1|} |\vec{\mathbf{d_s}}|, 1 \leq i, j \leq |V_1|. \tag{21}$$

For any $e =< v_i, v_j >\in E_1$ with annotation $\vec{\mathbf{d_k}}$, with the inequality (21), we have

$$|\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}| \leq |\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}}| + |\vec{\mathbf{d_k}}| \leq 2\Sigma_{s=1}^{|E_1|} |\vec{\mathbf{d_s}}|. \tag{22}$$

For the inequality (22), based on the inequality (12), we have

$$|\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}| < \vec{\beta}, e =< v_i, v_j >\in E_1 \text{ is annotated with } \vec{\mathbf{d_k}}. \tag{23}$$

LEMMA 7. *We have* $\vec{\sigma}(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) \geq 0$, *where* $e =< v_i, v_j >\in E_1$ *is annotated with* $\vec{\mathbf{d_k}}$, *subject to the constraints (8) and (23).*

PROOF. If $\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}} = \vec{\mathbf{0}}$, then $\vec{\sigma}(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) \geq 0$ holds.

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

With the constraint (23), we have

$$\begin{aligned}
&\vec{\sigma}(\vec{\mathbf{q_j}} - \vec{\mathbf{q_i}} + \vec{\mathbf{d_k}}) \\
&\geq \vec{\sigma}(0, \ldots, 0, q_j^{(h)} - q_i^{(h)} + d_k^{(h)}, -\beta^{(h+1)} + 1, \ldots, -\beta^{(n)} + 1) \\
&= \sigma^{(h)}(q_j^{(h)} - q_i^{(h)} + d_k^{(h)}) - \sigma^{(h+1)}\beta^{(h+1)} + \sigma^{(h+1)} \\
&\quad - \ldots - \sigma^{(n)}\beta^{(n)} + \sigma^{(n)} \\
&= \sigma^{(h)}(q_j^{(h)} - q_i^{(h)} + d_k^{(h)} - 1) + \sigma^{(n)} \\
&> 0 \quad \square
\end{aligned}$$

Hence, Inequality (12) guarantees that the constraint (9) always holds when the optimality of **Problem 4** is achieved. The optimal solution for **Problem 4** is also an optimal solution for **Problem 3**.

3.6 Successive Shortest Path Algorithm

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

The algorithm is depicted in Figure 6. We let $f(e_k) = f'(e_k)\vec{\sigma}$ and $w(v_i) = w'(v_i)\vec{\sigma}$, where $f'(e_k)$ and $w'(v_i)$ are scalars. After the first **while** iteration, the algorithm always

Input: $G_1 = (V_1, E_1)$
Output: $\vec{\mathbf{q_i}}, 1 \leq i \leq |V_1|$
Procedure:
  $f'(e_k) = 0$ for $1 \leq k \leq |E_1|$, $\vec{\mathbf{q_i}} = \vec{\mathbf{0}}$ for $1 \leq i \leq |V_1|$.
  $e(v_i) = w'(v_i)$ for $1 \leq i \leq |V_1|$.
  Initialize the sets $E = \{v_i | e(v_i) < 0\}$ and $D = \{v_i | e(v_i) > 0\}$.
  while $(E \neq \phi)$ **do**
    Select a node $v_k \in E$ and $v_l \in D$.
    Determine shortest path distances $\vec{\kappa_{\mathbf{j}}}$ from node v_k to all other nodes in G_1 with respect to the residue costs $c_{ij}^{\pi} = \vec{\mathbf{d_{ij}}} - \vec{\mathbf{q_i}} + \vec{\mathbf{q_j}}$, where the edge $< v_i, v_j >$ is annotated with $\vec{\mathbf{d_{ij}}}$ in G_1.
    Let P denote a shortest path from v_k to v_l.
    Update $\vec{\mathbf{q_i}} = \vec{\mathbf{q_i}} - \vec{\kappa_{\mathbf{i}}}, 1 \leq i \leq |V_1|$.
    $\delta = min(-e(v_k), e(v_l), min\{r_{ij} | < v_i, v_j >\in P\})$, where r_{ij} is the flow value in the residue network flow graph.
    Augment δ units of flow along the path P.
    Update $f'(e_k)$, E, D, c_{ij} and the residue graph.
  end while

Figure 6: The successive shortest path algorithm

maintains feasible shifting factors and nonnegativity of flow values by satisfying the constraints (8) and (20). It adjusts the flow values such that the constraint (19) 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 Example 2 (in Figure 3), after applying *successive shortest path algorithm*, we have $\vec{\mathbf{p}}(L_1) = \vec{\mathbf{p}}(L_3) = \vec{\mathbf{p}}(L_4) = (1, 0)$ and $\vec{\mathbf{p}}(L_2) = (0, 0)$. Figure 7 shows the transformed code for Example 2 after memory reduction.

4. REFINEMENTS

4.1 Controlled Fusion

Although array contraction after loop fusion will decrease the overall memory requirement, loop fusion at too many loop levels 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 performing loop fusion at too many loop levels if the corresponding loops are shifted. This is because, after loop shifting, loop fusion at too many loop levels can potentially increase the number of operations either due to the IF-statements added to 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 still carefully

```
REAL*8 ZA(2:KN), za0, za1, ZB0(2:JN), ZB1(2:JN), zb
DO J = 2, JN
S1 : ZB1(J) = ZQ(J-1,2) + ZZ(J,2)
END DO
DO K = 3, KN
  DO J = 2, JN
    za1 = ZP(J-1,K) + ZR(J-1,K-2)
    zb = ZQ(J-1,K) + ZZ(J,K)
    IF (J.EQ.2) THEN
      ZP(J,K-1) = ZP(J,K-1) + za1 - ZA(K-1)
        -ZB1(J) + zb
      ZQ(J,K-1) = ZQ(J,K-1) + za1 + ZA(K-1)
        +ZB1(J) + zb
    ELSE
      ZP(J,K-1) = ZP(J,K-1) + za1 - za0
        -ZB1(J) + zb
      ZQ(J,K-1) = ZQ(J,K-1) + za1 + za0
        +ZB1(J) + zb
    END IF
S2 : ZB1(J) = zb
S3 : za0 = za1
  END DO
END DO
DO J = 2, JN
  za1 = (ZP(J-1,KN+1) + ZR(J-1,KN-1)
  IF (J.EQ.2) THEN
    ZP(J,KN) = ZP(J,KN) + za1 - ZA(KN)
      -ZB1(J) + ZB0(J)
    ZQ(J,KN) = ZQ(J,KN) + za1 + ZA(KN)
      ZB1(J) + ZB0(J)
  ELSE
    ZP(J,KN) = ZP(J,KN) + za1 - za0
      -ZB1(J) + ZB0(J)
    ZQ(J,KN) = ZQ(J,KN) + za1 + za0
      ZB1(J) + ZB0(J)
  END IF
S4 : za0 = za1
END DO
```

Figure 7: The transformed code for Figure 3(a) after memory reduction

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 [26] 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* [26]. 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.

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 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 SGI Origin 2000 multiprocessor. We present the experimental results on the R10K. The results on the Ultra II are similar [27]. The R10K processor has a 32KB 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.

Procedure Pick_and_Reject
Input: (1) a collection of m loop nests, (2) $\vec{\sigma}\vec{\mathbf{M}}_{\mathbf{i,k}}$ ($1 \le i \le m, 1 \le k \le \tau_i$), (3) the estimated number of register spills np and the estimated number of cache misses nm, both in the original loop nests.
Output: A set of loop nests to be fused, FS.
Procedure:

1. Initialize FS to be empty. Let OS initially contain all the m loop nests.
2. If OS is empty, return FS. Otherwise, select a loop nest L_i from OS such that the local array regions R written in L_i can be reduced most, i.e., the difference between the size of R and the number of the simultaneously live array elements due to the static write references in L_i, should lexically be neither smaller than nor equal to any other loop nest in OS. Let TR be the set of loop nests in OS which contain references to R. Estimate a, the number of register spills, and b, the number of cache misses, after fusing the loops in both FS and TR and after performing array contraction for the fused loop. If ($a \le np \wedge b \le nm$), then $FS \leftarrow FS \cup TR$, $OS \leftarrow OS - TR$. Otherwise, $OS \leftarrow OS - \{L_i\}$ and go to step 2.

Figure 8: Procedure Pick_and_Reject

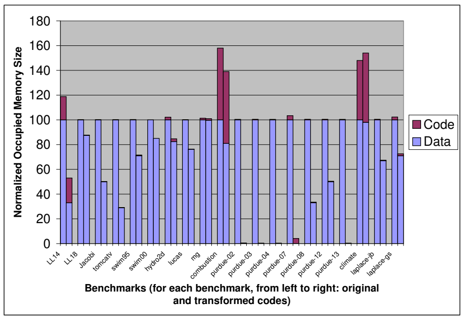
5.1 Benchmarks and Memory Reduction

Table 1 lists the benchmarks used in our experiments, their descriptions and their input parameters. These benchmarks are chosen because they either readily fit our program model or they can be transformed by our enabling algorithms to fit. With additional enabling algorithms developed in the future, we hope to collect more test programs. In this table, "m/n" 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 benchmark `swim` from SPEC95 and SPEC2000, we denote the SPEC95 version as `swim95` and the SPEC2000 version as `swim00`. Program `swim00` is almost identical to `swim95` 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 `climate`, `laplace-jb`, `laplace-gs` and all the Purdue set problems are from an HPF benchmark suite at Rice University [20, 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 the 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 `swim95`, `swim00` 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 contrac-

Table 1: Test programs

Benchmark Name	Description	Input Parameters	m/n
LL14	Livermore Loop No. 14	N = 1001, ITMAX = 50000	3/1
LL18	Livermore Loop No. 18	N = 400, ITMAX = 100	3/2
Jacobi	Jacobi Kernel w/o convergence test	N = 1100, ITMAX = 1050	2/2
tomcatv	A mesh generation program from SPEC95fp	reference input	5/1
swim95	A weather prediction program from SPEC95fp	reference input	2/2
swim00	A weather prediction program from SPEC2000fp	reference input	2/2
hydro2d	An astrophysical program from SPEC95fp	reference input	10/2
lucas	A promality test from SPEC2000fp	reference input	3/1
mg	A multigrid solver from NPB2.3-serial benchmark	Class 'W'	2/1
combustion	A thermochemical program from UMD Chaos group	N1 = 10, N2 = 10	1/2
purdue-02	Purdue set problem02	reference input	2/1
purdue-03	Purdue set problem03	reference input	3/2
purdue-04	Purdue set problem04	reference input	3/2
purdue-07	Purdue set problem07	reference input	1/2
purdue-08	Purdue set problem08	reference input	1/2
purdue-12	Purdue set problem12	reference input	4/2
purdue-13	Purdue set problem13	reference input	2/1
climate	A two-layer shallow water climate model from Rice	reference input	2/4
laplace-jb	Jacobi method of Laplace from Rice	ICYCLE = 500	4/2
laplace-gs	Gauss-Seidel method of Laplace from Rice	ICYCLE = 500	3/2



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

LL14	LL18	Jacobi	tomcatv	swim95
96	11520	19360	14750	14794
swim00	hydro2d	lucas	mg	combustion
191000	11405	142000	8300	89
purdue-02	purdue-03	purdue-04	purdue-07	purdue-08
4198	4198	4194	524	4729
purdue-12	purdue-13	climate	laplace-jb	laplace-gs
4194	4194	169	6292	1864

Figure 9: Memory sizes before and after transformation

tion. Among these programs, only `combustion, purdue-07 and purdue-08` fit the program model in [8]. In those cases, the algorithm in [8] will derive the same result as ours. So, there is no need to list those results.

For all versions of the benchmarks, we use the native Fortran compilers to produce the machine codes. We simply use the optimization flag "-O3" with the following adjustments. We switch off prefetching for `laplace-jb`, software pipelining for `laplace-gs` and loop unrolling for `purdue-03`. For `swim95` and `swim00`, 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. We compute the data size based on the global data in common blocks and the local data defined in the main program. 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. 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 considerably. The arithmetic mean of the reduction rate, counting both the data and the code, is 51% for all benchmarks. For several small `purdue` benchmarks, the reduction rate is almost 100%.

5.2 Performance

Figure 10 compares the normalized execution time, 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. The geometric mean of speedup after memory reduction is 1.40 over all benchmarks.

The best speedup of 5.67 is achieved for program `purdue-03`.

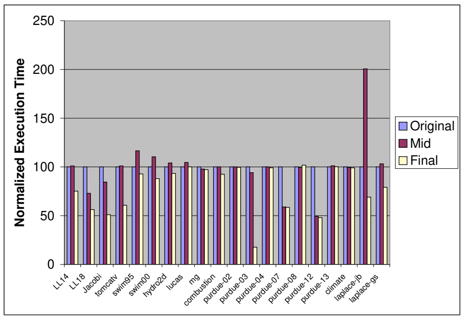
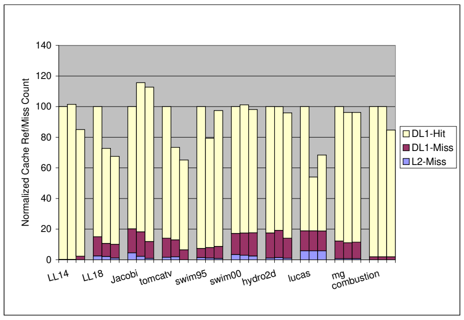


Figure 10: Performance before and after transformation



(Original, Mid and Final are from left to right for each benchmark)

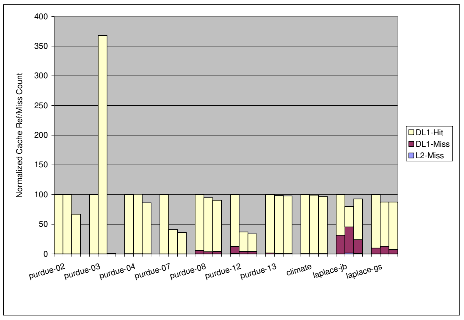
Figure 11: Cache statistics before and after transformation

This program 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.

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. We use the **perfex** package to get the cache statistics. Figures 11 and 12 compare such statistics, where the total reference counts in the original codes are normalized to 100.

When arrays are contracted to scalars, register reuse is often increased. Figures 11 and 12 show that the number of total references get decreased in most of the cases. The total number of reference counts, over all benchmarks, is reduced by 21.1%. However, in a few cases, the total reference counts get increased instead. We examined the assembly codes and found a number of reasons:



(Original, Mid and Final are from left to right for each benchmark)

Figure 12: Cache statistics before and after transformation (cont.)

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 decease 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 11 and 12 show that cache misses are generally reduced by memory reduction. The total number of cache misses, over all benchmarks, is reduced by 63.8% after memory reduction. The total number of L1 data cache misses is reduced by 57.3% after memory reduction. The improved cache performance seems to often have a bigger impact on execution time than the total reference counts.

5.4 Other Experiments

In [27], we reported how our memory reduction technique affects prefetching, software pipelining, register allocation and unroll-and-jam. We conclude that our technique does not seem to create difficulties for these optimizations.

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 *retiming* [16] to adjust the iteration space for individual statements such that the total buffer size can be minimized. We have compared their algorithm with ours in the introduction and in Section 5.1.

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 by 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 [15]. Their program model does not allow fusion-preventing dependences. Lewis *et al.* proposes to apply loop fusion and array contraction directly to array statements for array languages such as F90 [17]. The same result can be achieved if the array statements are transformed into various loops and loop fusion and array contraction are then applied. 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 shifting and does not provide formal algorithms and evaluations.

Loop fusion has been studied extensively. To name a few publications, Kennedy and McKinley prove maximizing data locality by loop fusion is NP-hard [13]. They provide two polynomial-time heuristics. Singhai and McKinley present *parameterized loop fusion* to improve parallelism and cache locality [25]. They do not perform memory reduction or loop shifting. Recently, Darte analyzes the complexity of loop fusions [5] and claims that the problem of maximum fusion of parallel loops with constant dependence distances is NP-complete when combined with loop shifting. His goal is to find the minimum number of partitions such that the loops within each partition can be fused, possibly enabled by loop shifting, and the fused loop remains parallel. Mainly because of different objective functions, his problem and ours yield completely different complexity. Manjikian and Abdelrahman present *shift-and-peel* [18]. They shift the loops in order to enable fusion. None of the works listed above address the issue of minimizing memory requirement for a collection of loops and their techniques are very different from ours.

7. CONCLUSION

In this paper, we propose to enhance data locality via a memory reduction technique, 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 in $O(|V|^3)$ time. 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 1.40 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. ACKNOWLEDGEMENTS

This work is sponsored in part by National Science Foundation through grants CCR-9975309, ACI/ITR-0082834 and MIP-9610379, by Indiana 21st Century Fund, by Purdue Research Foundation, and by a donation from Sun Microsystems, Inc.

9. REFERENCES

[1] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1993.

[2] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows.* Wiley, New York, 1990.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 53–65, White Plains, New York, June 1990.

[4] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, December 1996.

[5] A. Darte. On the complixity of loop fusion. In *Proceedings of International Conference on Parallel Architecture and Compilation Techniques*, pages 149–157, Newport Beach, California, October 1999.

[6] C. Ding. *Improving Effective Bandwidth Through Compiler Enhancement of Global and Dynamic Cache Reuse.* PhD thesis, Department of Computer Science, Rice University, January 2000.

[7] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, August 1991. Also in *Lecture Notes in Computer Science*, pp. 328-341, Springer-Verlag, August 1991.

[8] A. Fraboulet, G. Hurard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the Twelfth International Symposium on System Synthesis*, Boca Raton, Florida, November 1999.

[9] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[10] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*. Also in No. 757 in *Lecture Notes in Computer Science*, pages 281-295, Springer-Verlag, 1992.

[11] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software-Practice and Experience*, 20(2), February 1990.

[12] J. Gu, Z. Li, and G. Lee. Experience with efficient array data flow analysis for array privatization. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–167, Las Vegas, NV, June 1997.

[13] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Springer-Verlag Lecture Notes in Computer Science, 768. Proceedings of the Sixth Workhsop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August, 1993.

[14] D. J. Kuck. *The Structure of Computers and Computations*, volume 1. John Wiley & Sons, 1978.

[15] C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Optimization of memory usage and communication requirements for a class of loops implementing multi-dimensional integrals. In *Proceedings of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[16] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[17] E. C. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, Montreal, Canada, June 1998.

[18] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, February 1997.

[19] D. Maydan, S. Amarasinghe, and M. Lam. Array data-flow analysis and its use in array privatization. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, Charleston, SC, January 1993.

[20] A. G. Mohamed, G. C. Fox, G. von Laszewski, M. Parashar, T. Haupt, K. Mills, Y.-H. Lu, N.-T. Lin, and N.-K. Yeh. Applications benchmark set for fortran-d and high performance fortran. Technical Report CRPS-TR92260, Center for Research on Parallel Computation, Rice University, June 1992.

[21] J. Rice and J. Jing. Problems to test parallel and vector languages. Technical Report CSD-TR-1016, Department of Computer Science, Purdue University, 1990.

[22] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the ACM International Conference on Supercomputing*, pages 153–166, Santa Fe, NM, May 2000.

[23] V. Sarkar and G. R. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of 1991 ACM International Conference on Supercomputing*, pages 194–205, Cologne, Germany, June 1991.

[24] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[25] S. K. Singhai and K. S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6), 1997.

[26] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, GA, May 1999.

[27] Y. Song, R. Xu, C. Wang, and Z. Li. Performance enhancement by memory reduction. Technical Report CSD-TR-00-016, Department of Computer Science, Purdue University, 2000. Also available at http://www.cs.purdue.edu/homes/songyh/academic.html.

[28] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, San Jose, CA, October 1998.

[29] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, Stanford University, August 1992.

[30] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.