2000

Performance Enhancement by Memory Reduction

Yonghong Song

Rong Xu

Cheng Wang

Zhiyuan Li
Purdue University, li@cs.purdue.edu

Report Number:
00-016

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 examples, 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 ached data due to the reduced working set.

*Technical report CSD-TR-00-0016, Department of Computer Sciences, Purdue University, November, 2000.

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

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, ..., 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 $L_{i,1}$, $L_{i,2}$, ..., $L_{i,n}$, $n \geq 1$, 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

(a)
```
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
```

(b)
```
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
```

(c)
```
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
```

(d)
```
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
```

Figure 1: Example 1

(a)
```
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}
```

(b)
```
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)
```

(c)
```
DO L_1 = l_1, u_1
...
DO L_i = l_i, u_i
...
DO L_m = l_m, u_m
...
```

(d)
```
DO L = min_{i=1}^m l_i, max_{i=1}^m u_i
...
```

(e)
```
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))
```

Figure 2: The original and the transformed loop nests

$u_{i,j}$ respectively, where $l_{i,j}$ and $u_{i,j}$ are loop invariants. For simplicity of presentation, all the loop nests L_i, $1 \le i \le m$, 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 I_1 and I_2 and is 1 for loop I_3. We first apply our technique to fuse loops I_1, I_2 and I_3 at the outmost level only, resulting in the loop nest shown in Figure 3(b). We then apply the technique to loops J_1 and J_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_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, 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 $[JN+1, KN+1]$. *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, 26] [1] 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 Fig-

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

```
DO I1 = ...
    DO J1 = ...
       ...
    END DO
END DO
DO I2 = ...
    DO J1 = ...
       ...
    END DO
END DO
DO I3 = ...
   ...
END DO
```

(a)

```
DO I = ...
    DO J1 = ...
       ...
    END DO
    DO J2 = ...
       ...
    END DO
    ...
END DO
```

(b)

```
DO I = ...
    DO J = ...
       ...
    END DO
    ...
END DO
```

(c)

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

ure 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* [2] $\vec{\mathrm{dv}}(e)$.

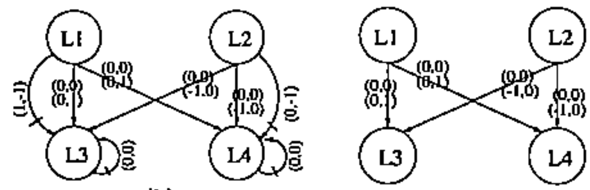
For each dependence edge e, if its distance vector is not constant, we replace it with a set of edges as follows. Let S be the set of dependence distances e represents. Let $\vec{\mathrm{d_0}}$ be the lexicographically minimum distance in S. Let $S_1 = \{\vec{\mathrm{d_1}} | \vec{\mathrm{d_1}} \not\le \vec{\mathrm{d}},\ \vec{\mathrm{d_1}} \in S \wedge (\forall \vec{\mathrm{d}} \in S \wedge \vec{\mathrm{d}} \neq \vec{\mathrm{d_1}})\}$. For any vector $\vec{\mathrm{d_1}}$ in S_1 (also in S), there exists no other vector in S which is no smaller than $\vec{\mathrm{d_1}}$. We replace the original edge e with $(|S_1| + 1)$ edges, annotated by $\vec{\mathrm{d_0}}$ and $\vec{\mathrm{d_i}}$ ($\vec{\mathrm{d_i}} \in S_1, 1 \le i \le |S_1|$) respectively.

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 $\vec{\mathrm{dv}} = (0, 0)$ is due to array region $ZA(2 : JN, 2 : KN)$. 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]From [29, 30], $\vec{\mathrm{u}} = (u_1, u_2, \ldots, u_n)$, $\vec{\mathrm{v}} = (v_1, v_2, \ldots, v_n)$, $\vec{\mathrm{u}} + \vec{\mathrm{v}} = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n)$, $\vec{\mathrm{u}} - \vec{\mathrm{v}} = (u_1 - v_1, u_2 - v_2, \ldots, u_n - v_n)$, $\vec{\mathrm{u}} \succ \vec{\mathrm{v}}$ ($\vec{\mathrm{u}}$ is lexicographically greater than $\vec{\mathrm{v}}$) if $\exists 0 \le k \le n-1, (u_1, \ldots, u_k) = (v_1, \ldots, v_k) \wedge u_{k+1} > v_{k+1}$, $\vec{\mathrm{u}} \succeq \vec{\mathrm{v}}$ if $\vec{\mathrm{u}} \succ \vec{\mathrm{v}}$ or $\vec{\mathrm{u}} = \vec{\mathrm{v}}$, $\vec{\mathrm{u}} \ge \vec{\mathrm{v}}$ if $u_k \ge v_k$ $(1 \le k \le n)$.

```
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 4: Example 2 and its original and simplified loop dependence graphs

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 $u_{i,h} - l_{i,h} + 1 = u_{j,h} - l_{j,h} + 1, 1 \le i, j \le m, 1 \le h \le n$.

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 symbolicly constant w.r.t. the program segment under consideration. Denote $\vec{\beta} = (\beta^{(1)}, \ldots, \beta^{(n)})$. We let $\sigma^{(n)} = 1$ and $\sigma^{(h)} = \sigma^{(h+1)}\beta^{(h+1)}, 1 \le h \le n-1$. 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 [3] 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 4(b) as an example, we have $\vec{\beta} = (KN - 1, JN - 1)$, $\vec{\sigma} = (JN - 1, 1)$, $\tau_1 = \tau_2 = 1$ and $\tau_3 = \tau_4 = 0$.

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

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.

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, $i \neq j$, 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 $\tau_i > 0$, 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.
- Based on the claims 1 and 3, for each node L_i such that $\tau_i = 0$, we remove all dependence edges from L_i to itself.
- Based on the claims 2 and 3, 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.
- Based on the claims 2 and 3, 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 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_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 $ZA(J, K)$.

2.5 Reference Windows

Loop shifting is applied before loop fusion in order to honor all the dependences. We associate one integer vector $\vec{p}(L_i)$ with each loop nest L_i in the loop dependence graph. Denote $\vec{p}(L_j) = (p^1(L_j), \ldots, p^n(L_j))$ 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 $\vec{dv}$, the new distance vector is $\vec{p}(L_j) + \vec{dv} - \vec{p}(L_i)$. 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.

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, ..., e_s, suppose their reference windows at time t are W_1, W_2, ..., 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 $(\bar{j}_1, \ldots, \bar{j}_n)$ 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.* □

In this paper, $\vec{u}\vec{v}$ denotes the inner product of $\vec{u}$ and $\vec{v}$. Given a dependence π with the distance vector $\vec{dv} = (d^1, d^2, \ldots, d^m)$ after loop shifting, $\vec{\sigma}\vec{dv}$ 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, $\vec{\sigma}\vec{dv}$ 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.* □

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

$$\vec{dv}_1 = \vec{p}(L_3) + (-1,0) - \vec{p}(L_2) \succeq \vec{0} \quad (1)$$

$$\vec{dv}_2 = \vec{p}(L_4) + (-1,0) - \vec{p}(L_2) \succeq \vec{0} \quad (2)$$

$$\vec{dv}_3 = \vec{p}(L_3) + (0,0) - \vec{p}(L_2) \succeq \vec{0} \quad (3)$$

$$\vec{dv}_4 = \vec{p}(L_4) + (0,0) - \vec{p}(L_2) \succeq \vec{0} \quad (4)$$

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

$$\vec{\sigma}\vec{dv}_i \geq 0, 1 \leq i \leq 4 \quad (5)$$

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

$\vec{\sigma}\vec{dv}_3$ represents the predicted reference window size for the flow dependence from L_2 to L_3, and $\vec{\sigma}\vec{dv}_4$ for the predicted reference window size for the flow dependence from L_2 to L_4. The size of the predicted reference window of L_2 can be computed by taking the greater one of the above two reference window sizes, i.e., $max(\vec{\sigma}\vec{\mathbf{dv_3}}, \vec{\sigma}\vec{\mathbf{dv_4}})$, 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.)

$$min(\Sigma_{i=1}^{m}\Sigma_{k=1}^{\tau_i}\vec{\sigma}\vec{\mathbf{M_{i,k}}}) \tag{6}$$

subject to

$$\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)) \geq 0, \forall \text{ L-edge } e \tag{7}$$

$$\vec{\sigma}\vec{\mathbf{M_{i,k}}} \geq \vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)), \forall \text{ M-edge } e, 1 \leq k \leq \tau_i \tag{8}$$

We call the above defined system as **Problem 1**. In (6), $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$ represents the predicted reference window size for the local array regions due to the kth 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, $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$, must be no smaller than the predicted reference window size for every M-edge originated from L_i and due to the kth 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, $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i)) \geq 0$ 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 $\vec{\mathbf{dv}}(e) = \vec{0}$, 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 (7) 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 $\vec{\mathbf{dv}}(e_2) - \vec{\mathbf{dv}}(e_1) \succeq \vec{0}$. 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 2.4, 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, $\vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_4) - \vec{\mathbf{p}}(L_i)) \geq \vec{\sigma}(\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e_3) - \vec{\mathbf{p}}(L_i))$ holds. Hence, the constraint (8) 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 (8) correctly computes the predicted reference window size, $\vec{\sigma}\vec{\mathbf{M_{i,k}}}$.

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.)

$$\vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i) \succeq \vec{0}, \forall \text{ L-edge } e \tag{9}$$

$$\vec{\mathbf{M_{i,k}}} \succeq \vec{\mathbf{p}}(L_j) + \vec{\mathbf{dv}}(e) - \vec{\mathbf{p}}(L_i), \forall \text{ M-edge } e, 1 \leq k \leq \tau_i \tag{10}$$

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 LDG 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 ($\vec{\mathbf{p}}$) 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 (6) is not changed.*
2. *If all $\vec{\mathbf{p}}$ 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 $\vec{dv}' = \vec{p}(L_j) + \vec{dv}(e) - \vec{p}(L_i) = (0, \ldots, 0, c_1, \ldots)$ where $c_1 < 0$ is the sth and the first nonzero component of $\vec{dv}'$. 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 $\vec{p}(L_j)$ by $\vec{p}(L_j) = \vec{p}(L_j) + \vec{\delta}$. Because of $\vec{\sigma}\vec{\delta} = 0$, the new $\vec{p}$ values, including $\vec{p}(L_j)$, satisfy the constraints (7) and (8). The value for the objective function (6) is also not changed.

If $\vec{p}(L_j) + \vec{dv}(e) - \vec{p}(L_i)$ 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 $\vec{p}(L_j)$ is increased compared to its original value. For any L-edge with the destination node L_j, if the constraint (9) holds before updating $\vec{p}(L_j)$, it still holds after the update. Such a property will guarantee our process to terminate.

Go to step 2.

4. *This step finds $\vec{M_{l,k}}$ values which satisfy the constraint (10).*

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

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

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

With such $\vec{p}$ and $\vec{M_{l,k}}$ 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).* □

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 $\vec{M_{l,k}}$ values as the solution for* **Problem 1**. *Such $\vec{p}$ and $\vec{M_{l,k}}$ 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**. □

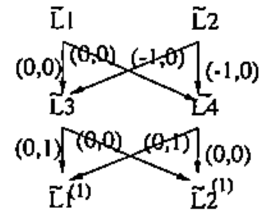


Figure 5: The transformed graph (G_1) for Figure 4(c)

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 $\tilde{L_i}$ 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 ($1 \leq k \leq \tau_i$) 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 $\vec{0}$.
- For any M-edge $< L_i, L_j >$ in G due to the static write reference r_k, suppose its distance vector $\vec{dv}$. Add an edge $< \tilde{L_j}, \tilde{L_i}^{(k)} >$ to G_1 with the distance vector $-\vec{dv}$.
- For any L-edge $< L_i, L_j >$ in G, suppose its distance vector $\vec{dv}$. Add an edge $< \tilde{L_i}, \tilde{L_j} >$ to G_1 with the distance vector $\vec{dv}$.

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

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

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

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 $\vec{d_k}$.)

$$min\Sigma_{i=1}^{|V_1|}w(v_i)\vec{q_i} \quad (11)$$

subject to

$$\vec{q_j} - \vec{q_i} + \vec{d_k} \succeq \vec{0}, \forall e \quad (12)$$

$$\vec{\sigma}(\vec{q_j} - \vec{q_i} + \vec{d_k}) \geq 0, \forall e \quad (13)$$

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

PROOF. *We have*

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

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

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

Hence the objective function (6) is equivalent to (11).

For each edge $e =< \bar{L_i}, \bar{L_j} >$ *in* G_1*, the inequality (12) is equivalent to*

$$\vec{p}(L_j) - \vec{p}(L_i) + \vec{dv}(e_1) \succeq \vec{0}, \quad (14)$$

where e_1 *is an L-edge in* G *from* L_i *to* L_j*. Inequality (14) is equivalent to (9), hence inequality (12) is equivalent to (9).*

For each edge $e =< \bar{L_j}, \bar{L_i}^{(k)} >$ *in* G_1*, the inequality (12) is equivalent to*

$$\vec{M_{i,k}} + \vec{p}(L_i) - \vec{p}(L_j) - \vec{dv}(e_1) \succeq \vec{0}, \quad (15)$$

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 (15) is equivalent to (10), hence inequality (12) is equivalent to (10).*

Similarly, it is easy to show that the constraints (7) and (8) are equivalent to constraint (13). □

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

$$\Sigma_{k=1}^{|E_1|}|\vec{dv}(e_k)| < \frac{1}{2}\vec{\beta}, \quad (16)$$

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

If we consider the vector as the basic computation unit, **Problem 3** is a nonlinear system, due to the constraint (13). The same as **Problem 2**, such a nonlinear system 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, however, we show that we can achieve an optimal solution in polynomial time for **Problem 3** by utilizing the network flow property.

3.3 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 each 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,

$$\Sigma_{e_k=<.,v_i>\in E_1}f(e_k) = w(v_i) + \Sigma_{e_k=<v_i,.>\in E_1}f(e_k), \quad (17)$$

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*.* □

Based on equation (17), given a legal network flow, we have

$$\Sigma_{i=1}^{|V_1|}w(v_i)\vec{q_i} = \Sigma_{k=1}^{|E_1|}f(e_k)(\vec{q_j} - \vec{q_i}) \quad (18)$$

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 (13), we have

$$f(e_k)(\vec{q_j} - \vec{q_i} + \vec{d_k}) = c_k\vec{\sigma}(\vec{q_j} - \vec{q_i} + \vec{d_k}) \geq 0. \quad (19)$$

Hence, we have

$$f(e_k)(\vec{q_j} - \vec{q_i}) \geq -f(e_k)\vec{d_k}. \quad (20)$$

Therefore, with the equation (18), if $f(e_k) \succeq \vec{0}$, we have

$$\Sigma_{i=1}^{|V_1|}w(v_i)\vec{q_i} \geq -\Sigma_{k=1}^{|E_1|}f(e_k)\vec{d_k}. \quad (21)$$

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 $f(e_k) = c_k\vec{\sigma}$ exists such that $c_k \geq 0$ for $1 \leq k \leq |E_1|$, and*

3. $\Sigma_{i=1}^{|V_1|} w(v_i)\vec{q_i} = -\Sigma_{k=1}^{|E_1|} f(e_k)\vec{d_k}$ *holds, i.e., inequality (21) becomes an equality.*

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

PROOF. *Obvious from the above discussion.* □

3.4 Solving Problem 3

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

$$max_{k=1}^{|E_1|}(-f(e_k)\vec{d_k}) \tag{22}$$

subject to

$$\Sigma_{e_k=<.,v_i>\in E_1} f(e_k) = w(v_i) + \Sigma_{e_k=<v_i,.>\in E_1} f(e_k), 1 \leq i \leq |V_1|. \tag{23}$$

$$f(e_i) \succeq \vec{0}, 1 \leq i \leq |E_1|. \tag{24}$$

The constraint (13) is mandatory for the equivalence between **Problem 3** and its dual problem, following the development of optimality conditions in Section 3.3 [1]. The constraint (23) 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 (11)-(12) and (22)-(24). 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 (12), (23) and (24) holds, and
- The objective function values for (11) and (22) are equal, i.e., $\Sigma_{i=1}^{|V_1|} w(v_i)\vec{q_i} = -\Sigma_{k=1}^{|E_1|} f(e_k)\vec{d_k}$ 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 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. 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 $O(|V_1|^3)$, *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 $\vec{q_i}$ in* **Problem 4**, *there exists a spanning tree T in G_1 such that each edge $e =< v_i, v_j >$ in T satisfies $\vec{q_j} - \vec{q_i} + \vec{d_k} = \vec{0}$.*

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 $\vec{q}$ to be $\vec{0}$, all $\vec{q_i}, 1 \leq i \leq |V_1|$, can be determined uniquely. With such uniquely-determined $\vec{q_i}$, we have

$$|\vec{q_j} - \vec{q_i}| \leq \Sigma_{s=1}^{|E_1|}|\vec{d_s}|, 1 \leq i, j \leq |V_1|. \tag{25}$$

For any $e =< v_i, v_j >\in E_1$ with annotation $\vec{d_k}$, with the inequality (25), we have

$$|\vec{q_j} - \vec{q_i} + \vec{d_k}| \leq |\vec{q_j} - \vec{q_i}| + |\vec{d_k}| \leq 2\Sigma_{s=1}^{|E_1|}|\vec{d_s}|. \tag{26}$$

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

$$|\vec{q_j} - \vec{q_i} + \vec{d_k}| < \vec{\beta}, e =< v_i, v_j >\in E_1 \text{ is annotated with } \vec{d_k}. \tag{27}$$

LEMMA 7. *$\vec{\sigma}(\vec{q_j} - \vec{q_i} + \vec{d_k}) \geq 0$, where $e =< v_i, v_j >\in E_1$ is annotated with $\vec{d_k}$, subject to the constraints (12) and (27).*

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

Otherwise, assume the first non-zero component is the hth for $\vec{q_j} - \vec{q_i} + \vec{d_k}$. 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 (27), we have

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

Hence, Inequality (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**.

Input: $G_1 = (V_1, E_1)$
Output: $\vec{q_i}, 1 \le i \le |V_1|$
Procedure:
  $f'(e_k) = 0$ for $1 \le k \le |E_1|$, $\vec{q_i} = \vec{0}$ for $1 \le i \le |V_1|$.
  $e(v_i) = w'(v_i)$ for $1 \le i \le |V_1|$.
  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 $\vec{\kappa_j}$ from node v_k to all other nodes in G_1 with respect to the residue costs $c^{\pi}_{ij} = \vec{d_{ij}} - \vec{q_i} + \vec{q_j}$, where the edge $< v_i, v_j >$ is annotated with $\vec{d_{ij}}$ in G_1.
    Let P denote a shortest path from v_k to v_l.
    Update $\vec{q_i} = \vec{q_i} - \vec{\kappa_i}, 1 \le i \le |V_1|$.
    $\delta = min(-e(v_k), e(v_l), min\{r_{ij} | < v_i, v_j > \in P\})$, 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

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 $f(e_k) = f'(e_k)\vec{\sigma}$ 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 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 [26] for details.

```
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 4(a) after memory reduction

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 at 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-

Procedure Pick_and_Reject
Input: (1) a collection of m loop nests, (2) $\partial \vec{M}_{i,k}$ $(1 \le i \le m, 1 \le k \le r_i)$, (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 size of the predicted reference window for L_i is no smaller than that for 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

print is the same in every T iteration. It is possible then to perform tiling on the whole T loop nest so as to exploit temporal locality across different T iterations [27]. On the other hand, after loop shifting plus fusion, the T loop and the fused loops form a $(n+1)$-level perfectly-nested loop nest. This resulting loop nest would appear to be a perfect candidate for tiling, since many tiling algorithms apply to perfectly-nested loops only. *However, the kind of shifting required for memory reduction often introduces very long backward dependences, which actually prevents profitable tiling.* (On ther other hand, partial memory reduction, which may not minimize the memory requirement, may allow profitable tiling. The interaction between partial memory reduction and tiling seems an interesting topic for our future research.) Where tiling and memory reduction can be performed separately, but not simultaneously, we need to make a choice, and we do so based on simple estimations of the cache miss penalty.

Let C_{b1} be the L1-cache line size and C_{b2} be the L2-cache line size, both measured in the number of data elements. Let p_1 be the L1-cache miss penalty and p_2 be the L2-cache miss penalty. Further, let W represent the footprint of the original loop body (in the number of data elements). We estimate the average cache miss penalty for each T-iteration in the original code by

$$\frac{p_1 W}{C_{b1}} + \frac{p_2 W}{C_{b2}}. \tag{28}$$

Likewise, let W_1 represent the footprint of the fused loop body after memory reduction. We estimate the average cache miss penalty for each T-iteration in the fused code by

$$\frac{p_1 W_1}{C_{b1}} + \frac{p_2 W_1}{C_{b2}}. \tag{29}$$

Obviously, W_1 is expected to be smaller than W. Under extreme circumstances, W_1 may completely fit in a certain cache, say the L2 cache, then the estimation is revised to remove the miss penalty on that cache.

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

$$\frac{p_1 S_1 W_2}{C_{b1} B_1} + \frac{p_2 S_2 W_2}{C_{b2} B_2}. \tag{30}$$

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

$$\frac{p_1 W_2}{C_{b1}} + \frac{p_2 S_1 W_2}{C_{b2} B_1}. \tag{31}$$

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

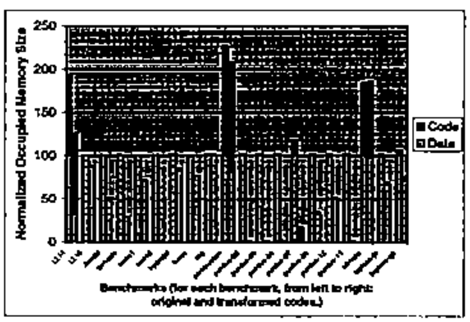
We have implemented our memory reduction technique in a research compiler, Panorama [12]. We implemented a network flow algorithm, *successive shortest path algorithm* [1]. The loop dependence graphs in our experiments are relatively simple. The *successive shortest path algorithm* takes less than 0.06 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 SGI Origin 2000 multiprocessor. The Ultra II processor has a 16KB 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 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.

5.1 Benchmarks and Memory Reduction

Table 1 lists the benchmarks used in our experiments, their descriptions and their input parameters. In the 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 two versions of `swim` in SPEC95 and SPEC2000, we call the SPEC95 version as `swim1` and the SPEC2000 version as `swim2`. `swim2` is almost identical to `swim1` 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,

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
swim1	A weather prediction program from SPEC95fp	reference input	2/2
swim2	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	swim1
96	11520	19360	14750	14794
swim2	hydro2d	lucas	mg	combustion
191000	11405	142000	8300	89
purdue-02	purdue-03	purdue-04	purdue-07	purdue-08
4198	4198	4194	524	4720
purdue-12	purdue-13	climate	laplace-jb	laplace-gs
4194	4194	169	6292	1864

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

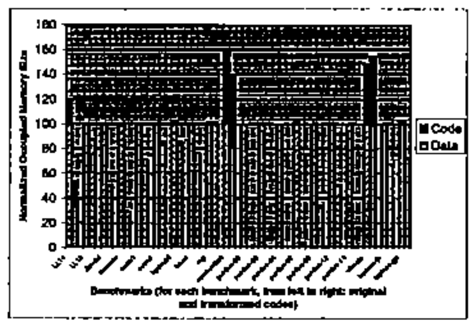


Figure 10: Memory sizes before and after transformation on the R10K

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 tomcatv 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

Figure 11: Performance before and after transformation on the Ultra II

Figure 12: Performance before and after transformation on the R10K

"-fast -xchip=ultra2 -xarch=v8plusa -fsimple=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-gs` and loop unrolling for `purdue-03`. For `swim1` and `swim2`, 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 running 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 is 5.67 on the R10K and is 41.3 on the Ultra II. 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. 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.

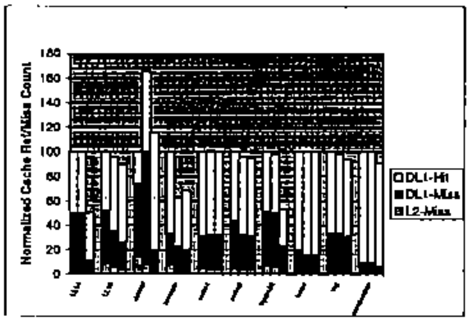
We see three programs actually get slowed down slightly after memory reduction. The execution time of both `purdue-13` and `laplace-gs` on the Ultra II is increased by 2%. The execution time of `purdue-08` on the R10K is increased by 1%. Both `purdue-08` and `purdue-13` make several math library function 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 *et al.* proposes 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

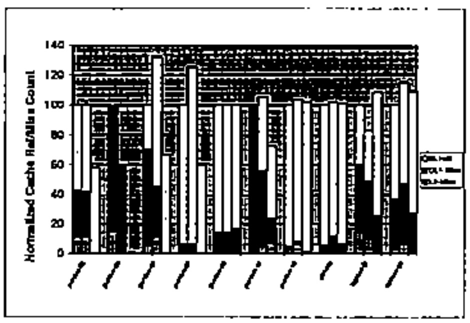
To further understand the effect of memory reduction on the performance, we examined the cache behavior of different versions of the tested benchmarks. We measured the reference count (dynamic load/store instructions), the miss count of the L1 data cache, and the miss count of the L2 unified cache on 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



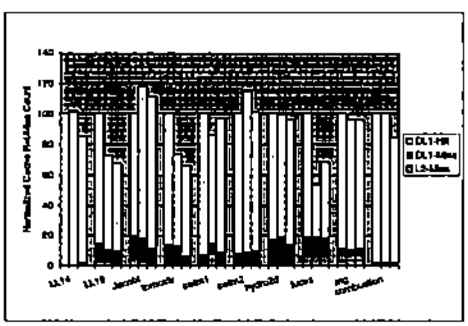
(Original, Mid and Final are from left to right for each benchmark)

Figure 13: Cache statistics before and after transformation on the Ultra II



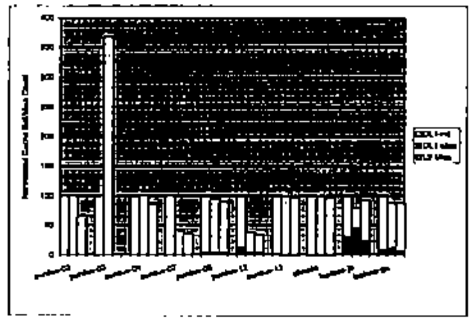
(Original, Mid and Final are from left to right for each benchmark)

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



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

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



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

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 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 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

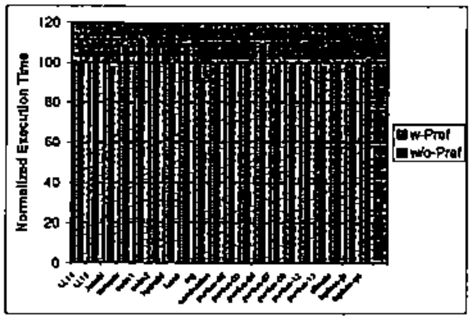


Figure 17: Performance of the original programs w/ and w/o prefetching on the Ultra II

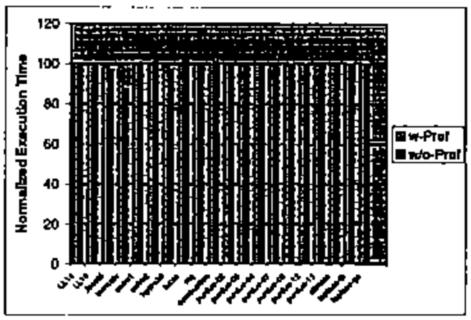


Figure 18: Performance of the transformed programs w/ and w/o prefetching on the Ultra II

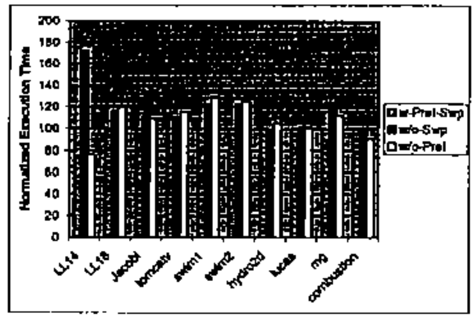


Figure 19: Performance of the original programs w/ and w/o prefetching and software pipelining on the R10K

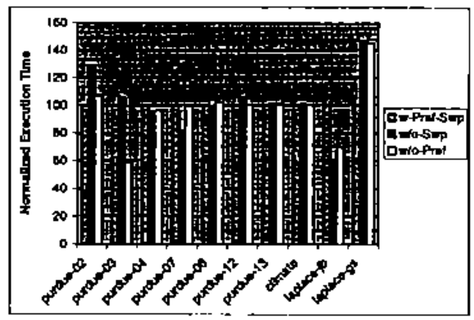


Figure 20: Performance of the original programs w/ and w/o prefetching and software pipelining on the R10K (cont.)

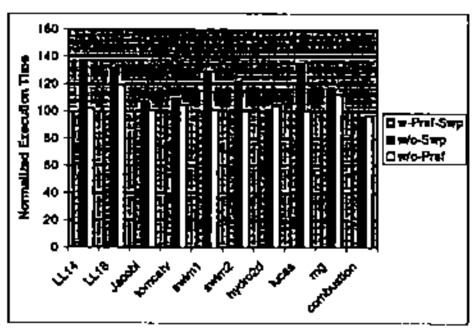


Figure 21: Performance of the transformed programs w/ and w/o prefetching and software pipelining on the R10K

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

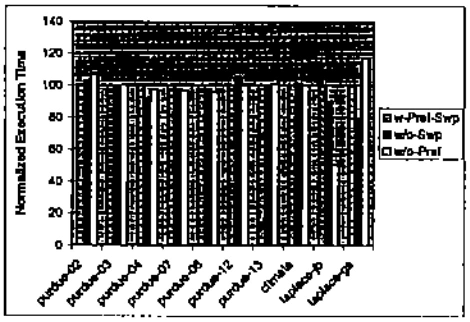


Figure 22: Performance of the transformed programs w/ and w/o prefetching and software pipelining on the R10K (cont.)

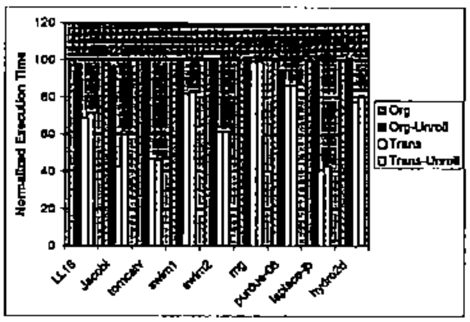


Figure 23: Performance of the code with (unroll-and-jam, scalar replcement) on the Ultra II

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 (536M 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. Figures 15 and 16 show that, in just two of the 20 codes transformed for the R10K, slightly more memory references are issued than

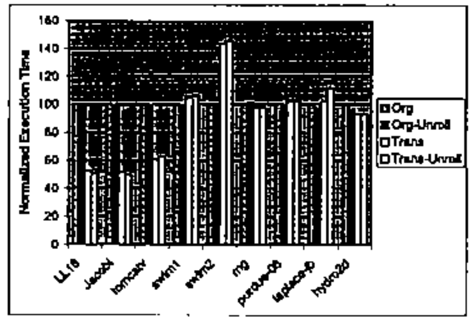


Figure 24: Performance of the code with (unroll-and-jam, scalar replacement) on the R10K

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 truely 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`, `tomcatv`, `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

Table 2: Performance of memory reduction vs. tiling (in seconds)

Benchmarks	Ultra II			R10K		
	Mem-Red	Tiling	$\frac{\text{Mem-Red}}{\text{Tiling}}$	Mem-Red	Tiling	$\frac{\text{Mem-Red}}{\text{Tiling}}$
LL18	8.3	7.5	1.11	4.79	5.67	0.84
Jacobi	88.2	32.9	2.20	68.58	39.64	1.73
tomcatv	80.1	96.0	0.83	70.26	72.73	0.97
swim1	118.4	74.5	1.59	86	58.50	1.47
swim2	863	790	1.09	645	508	1.27

memory reduction for **tomcatv**, also on both machines. This turns out to be correct in 9 out of the 10 cases. The exception is LL18 on the R10K. The tiled assembly code of LL18 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 [15]. 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 shifting and does not provide formal algorithms and evaluations. Gannon *et al.* introduce the concept of *reference window*, using it to estimate the cache hit rate and to guide program optimization for a software-controlled cache [9]. They do not address the memory reduction problem.

There exist a lot of work related with loop fusion. To name a few, 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. Megiddo and Sarkar use *mixed integer programming* to optimize weighted loop fusion for parallel programs [19]. 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. None of these works address the issue of minimizing memory requirement for a collection of loops and their techniques are very different from ours. Manjikian and Abdelrahman present *shift-and-peel* [17]. They shift the loops in order to enable fusion. However, they do not consider array contraction.

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 $O((|E|log|V|)(|E|+log|V|))$ 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

[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 4th International Workshop on Languages and Compilers for Parallel Computing*, August 1991. Also in *Lecture Notes in Computer Science*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds., pp. 328-341, Springer-Verlag, Aug. 1991.

[8] A. Fraboulet, G. Hurard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the 12th 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 *the fifth Workshop on Languages and Compilers for Parallel Computing*. Also in U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, 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 6th 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 *the twelfth International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[16] 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.

[17] 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.

[18] 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.

[19] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the ninth Annual ACM Symposium on Parallel Algorithms and Architecture*, pages 282–291, Newport, Rhode Island, USA, June 1997.

[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. *Compiler Algorithms for Efficient Use of Memory Systems*. PhD thesis, Department of Computer Sciences, Purdue University, November 2000.

[27] 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.

[28] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *Proceedings of the 8th 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.