# A Polynomial-Time Algorithm for Memory Space Reduction

Yonghong Song
Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
yonghong.song@sun.com

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

**Abstract**

Reducing memory-space requirement is important to many applications. For data-intensive applications, it may help avoid executing the program out-of-core. For high-performance computing, memory-space reduction may improve the cache hit rate as well as performance. For embedded systems, it can reduce the memory requirement, the memory latency and the energy consumption. This paper investigates program transformations which a compiler can use to reduce the memory space required for storing program data. In particular, the paper uses integer programming to model the problem of combining *loop shifting*, *loop fusion* and *array contraction* to minimize the data memory required to execute a collection of multi-level loop nests. The integer programming problem is then reduced to an equivalent network flow problem which can be solved in polynomial time.

**Key words:** compilers, optimization, graph theory, network flow problem

## 1 Introduction

Due to the speed gap between processors and memory, compiler techniques for efficient data access have been under extensive investigation. Many program transformations developed for program parallelization can be made useful for memory performance enhancement. In this paper, we investigate program transformations which can be performed by the compiler to reduce the memory space required for storing program data. In particular, we devise a network flow model to perform optimal *array contraction* in order to minimize the data memory required to execute a collection of multi-level loop nests. Reducing memory-space requirement is important in several areas. For high-performance computing, it may improve the cache hit rate as well as performance [1]. For data-intensive applications, it may help avoid executing the program out-of-core. For embedded systems, it can reduce the physical-memory requirement, the memory latency and the energy consumption [2, 3, 4].

The array contraction technique contracts multi-dimensional arrays to lower dimensions such that the total data size is reduced. The opportunities for array contraction exist often because the most natural way to specify a computational task may not be the most memory-efficient. Furthermore, 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 (*c.f.* Figure 1(b)), L1 and L2 can be fused (*c.f.* Figure 1(c)). Array $A$ can then be replaced by two scalars, $a1$ and $a2$, as Figure 1(d) shows. Later in this paper, our experimental data will show that such a combination of loop shifting, loop fusion, and array contraction is quite effective for memory-requirement reduction in a number of widely publicized benchmark programs. In this paper, we study how to optimally apply such a combination of program transforms.

In a related paper [1], we used a loop dependence graph to describe the data dependences which are critical to the legality and the optimality of array contraction. To build such a graph, it takes time that is exponential in terms of the depth of loop nesting. Fortunately, the loop nesting in practice is usually not very deep. We also showed how to formulate the optimal array contraction problem as an integer programming problem. In this paper, we shall develop a polynomial-time algorithm to solve such an integer programming problem. We also address a number of issues concerning code generation.

In the rest of the paper, we first describe our program model (in Section 2). We then formulate an integer programming problem (Section 3) and reduce it to a network flow problem (Section 4) which can be solved in polynomial time. Afterwards, we discuss code generation issues (Section 5) and experimental results (Section 6). We then compare with related work (Section 7) and draw a conclusion (Section 8).

## 2 Problem Model and Loop Dependence Graph

We consider a collection of loop nests, $L_1$, $L_2$, …, $L_m$, $m \geq 1$, as shown in Figure 2(a). Each label $L_i$ denotes a tight nest of loops with indices $L_{i,1}$, $L_{i,2}$, …, $L_{i,n}$, $n \geq 1$, listed from the outermost level to the innermost. (In the example in Figure 1(a), we have $m = 2$ and $n = 1$.) Loop $L_{i,j}$ has the lower bound $l_{ij}$ and the upper bound $l_{ij} + b_j - 1$ respectively, where $l_{ij}$ and $b_j$ are loop invariants. As in previous related work, all loops at the same level, $j$, are assumed to have the same trip count $b_j$. For convenience, we use the notation $\vec{b} = (b_1, b_2, \ldots, b_n)$ throughout the paper. We assume that none of the given loops can be partitioned into smaller loops by *loop distribution* [5]. (Otherwise, we first apply maximum loop distribution [5] to the given collection of loops.) It is only for the convenience of discussion that we require each loop nest $L_i$ to be tightly nested. If a loop nest $L_i$ is not tightly nested, we shall first apply the technique presented in this paper to its innermost loop body that consists of a collection of tightly-nested loops. After fusing such a collection of loops into a tight loop nest, we can work on the outer loop levels in $L_i$ in a similar way. Currently, our program model does not cover non-rectangular loop nests. We will study how to extend our program model in the future work.

The array regions referenced in the given collection of loops are divided into three categories. 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. Obviously, only the local array regions are amenable to array contraction. Local array regions can be recognized by applying existing dependence analysis, region analysis and live analysis techniques [6, 7, 8, 9], which will not be reviewed in this paper. In the example in Figure 1(a), $A(1:N)$ is the only local array region. Figure 3(a) shows a more complex example which resembles one of the well-known Livermore loops. In this example, where $m = 4$ and $n = 2$, each declared array is of dimension $[1 : JN + 1, 1 : KN + 1]$. *ZA(2:JN,2:KN)* and *ZB(2:JN,2:KN)* are local array regions.

To describe the dependence between a collection of loop nests, we extend the definitions of the traditional dependence distance vector and the dependence graph [10]. Following conventional notations [5, 11], given $\vec{u} = (u_1, u_2, \ldots, u_n)$ and $\vec{v} = (v_1, v_2, \ldots, v_n)$, we write $\vec{u} + \vec{v} = (u_1 + v_1, u_2 + v_2, \ldots, u_n + v_n)$, $\vec{u} - \vec{v} = (u_1 - v_1, u_2 - v_2, \ldots, u_n - v_n)$, $\vec{u} \succ \vec{v}$ if $\exists 0 \leq k \leq n-1, (u_1, \ldots, u_k) = (v_1, \ldots, v_k) \wedge u_{k+1} > v_{k+1}$, $\vec{u} \succeq \vec{v}$ if $\vec{u} \succ \vec{v}$ or $\vec{u} = \vec{v}$, and

$\vec{u} > \vec{v}$ if $u_k > v_k$ $(1 \le k \le n)$. We use $abs(\vec{u})$ to denote $(|u_1|, |u_2|, \ldots, |u_n|)$. Using these notations, we introduce the following definitions.

**Definition 1** *Given a collection of loop nests, $L_1$, …, $L_m$, as in Figure 2(a), if a data dependence exists from the source iteration $\vec{i} = (i_1, i_2, \ldots, i_n)$ of loop $L_1$ to the destination iteration $\vec{j} = (j_1, j_2, \ldots, j_n)$ of loop $L_2$, we say the* distance vector *is $\vec{j} - \vec{i} = (j_1 - i_1, j_2 - i_2, \ldots, j_n - i_n)$ for this dependence.*

**Definition 2** *Given a collection of loop nests, $L_1$, $L_2$, …, $L_m$, as in Figure 2(a), a* loop dependence graph *(LDG) is a directed multi-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 from $L_i$ to $L_j$ 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 *$\vec{d_e}$.*

In this paper, we assume constant dependence distance vectors. Certain non-constant dependence distances can be replaced by constant ones [1]. If there exist non-constant dependences which cannot be replaced, our technique will not be applied.

Existing techniques [5] can be utilized to construct the LDG. Despite the exponential worst-case complexity, the determination of data dependences is quite efficient in practice. Since this issue has been well studied, we do not discuss it further in this paper. Figure 3(b) shows the loop dependence graph for the example in Figure 3(a). The array regions associated with the dependence edges can be inferred from the program, and they are omitted in the figure. For instance, the flow dependence from $L_1$ to $L_3$ with $\vec{d} = (0, 0)$ 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, all these dependences are represented by a single arc. All associated distance vectors are then marked on this single arc.

## 3 An Integer Programming Problem

In this section, we model the problem of optimal array contraction (given the collection of loop nests) as an integer programming problem. For this purpose, we first discuss the legality for loop fusion and the method of loop shifting to ensure the legality. We then discuss the minimum data size required to execute the collection of loop nests after loop shifting and loop fusion. To ease the understanding of the underlying idea, we take an intermediate step of program transformation, called *loop coalescing*, which coalesces a loop nest of multiple levels into an equivalent single loop. However, we shall show later that we do not need to actually coalesce the loops in order to obtain the optimal array contraction.

### 3.1 Loop Coalescing

Loop coalescing transforms a tight multi-level loop nest into a single loop. To analyze the effect of loop coalescing, we define the *coalescing vector* $\vec{s} = (s_1, s_2, \ldots, s_n)$, where $s_i$ represents the number of times the innermost loop body of $L_i$ is executed within each $L_{i,j}$ loop iteration. Thus, we have $s_n = 1$, $s_h = s_{h+1}b_{h+1}, 1 \leq h \leq n-1$. For the loops in Figure 3(a), we have $\vec{s} = (JN-1, 1)$.

Given any iteration vector $\vec{i}$ of loop nest $L_i$ in Figure 2(a), the corresponding loop iteration index after loop coalescing equals $\vec{i}\vec{s}^T$, the inner product of $\vec{i}$ and $\vec{s}$. We call this inner product the *coalesced index*. Suppose $\vec{i_1}$ and $\vec{i_2}$ are two different iterations in a tight loop nest. In order for loop coalescing to be legal, the following equivalence must hold,

$$\vec{i_2} \succ \vec{i_1} \Leftrightarrow \vec{i_2}\vec{s}^T > \vec{i_1}\vec{s}^T. \tag{1}$$

**Lemma 1** *$\forall \vec{u}$ s.t. abs$(\vec{u}) < \vec{b}$, we have $\vec{u} \succ \vec{0} \Leftrightarrow \vec{u}\vec{s}^T > 0$. Furthermore, $\forall \vec{u}$ s.t. abs$(\vec{u}) < (\vec{b} + (0, \ldots, 0, 1))$, we have $\vec{u} \succeq \vec{0} \Rightarrow \vec{u}\vec{s}^T \geq 0$.*

**Proof** See Appendix A. □

Since $abs(\vec{i_2} - \vec{i_1}) < \vec{b}$ holds, Lemma 1 immediately derives the equivalence property (1), which means that loop coalescing preserves the execution order of all loop iterations and that the mapping between $\vec{i}$ and its coalesced index is one-to-one. Furthermore, the coalesced index values form a consecutive sequence, because the difference between the upper limit and the lower limit, plus one, equals $(\vec{b} - \vec{1})\vec{s}^T + 1 = \Pi_{k=1}^{n} b_k$, which is the total number of instances of the innermost loop body before coalescing. Here, $\vec{1}$ represents a vector with all elements equal to 1. Given the coalesced index and the trip counts $\vec{b}$, we can recompute $\vec{i}$ using *modulo* and *divide* operations. (We omit the straightforward derivation details.)

## 3.2 Legality Condition for Loop Fusion

Loop fusion must not reverse the source and destination of any dependence [5]. Therefore we have the following lemma.

**Lemma 2** *It is legal to fuse the collection of $m$ loop nests in Figure 2(a) into a single loop nest if and only if*

$$\vec{d_e} \succeq \vec{0}, \forall e \in E. \tag{2}$$

□

If there exist data dependences which do not satisfy the condition stated above, we perform loop shifting to satisfy the condition. We apply loop shifting to a loop $L_{i,j}$ by increasing both the lower and the upper loop bounds by a *shifting factor* $p_{ij}$. Within the loop body, we decrement the corresponding loop index variable by $p_{ij}$. For each of the $m$ loop nests in Figure 2(a), say $L_j$, we use the *shifting vector* $\vec{p_j} = (p_{j1}, p_{j2}, \ldots, p_{jn})$ to represent the shifting factors at various loop levels. After applying a shifting vector to each of the given $m$ loop nests, the code takes the form shown in Figure 2(b). For any dependence edge $e$ from $L_i$ to $L_j$ with the distance vector $\vec{d_e} = \vec{j} - \vec{i}$ in Figure 2(a), the new dependence distance vector after loop shifting equals $(\vec{j} + \vec{p_j}) - (\vec{i} + \vec{p_i}) = \vec{p_j} - \vec{p_i} + \vec{d_e}$.

To ease the understanding of the problem formulation, we perform loop coalescing as an intermediate program transformation between loop shifting and loop fusion. As the result of loop coalescing, those $m$ loop nests in Figure 2(b) become $m$ single loops. For the data dependence with a distance vector $\vec{p_j} - \vec{p_i} + \vec{d_e}$, its *coalesced dependence distance* after loop coalescing equals $(\vec{p_j} - \vec{p_i} + \vec{d_e})\vec{s}^T$. Following Lemma 2, we immediately have the following lemma.

**Lemma 3** *All the $m$ loops after loop shifting and loop coalescing can be legally fused into a single loop if and only if*

$$(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T \geq 0, \forall e \in E \textit{ from } L_i \textit{ to } L_j. \tag{3}$$

The lower bound of the single fused loop equals the smallest lower bound among the $m$ coalesced loops, and the new upper bound equals the greatest upper bound among those loops. We insert proper guards (i.e. IF statements) in the fused loop to ensure that no extraneous loop iterations get executed.

## 3.3 Minimizing the Array Size

We now analyze the array size required to execute the single fused loop after loop shifting, loop coalescing and loop fusion. We make the following assumptions to simplify the discussion.

**Assumption 1** *For each reference $w$ which writes to a local array region, $w$ writes to a distinct array element in every iteration of the innermost loop body. All values written by $w$ are useful, meaning that each written value will be used by some read references in the given set of loop nests.*

Assumption 1 stated above implies that we exclude the cases in which $w$ may not be executed in certain iterations. For such cases, we may need to take the IF conditions into account in order to determine the memory requirement. We also exclude the cases in which a reference may write to local regions of an array in certain loop iterations and to non-local regions of the same array in other iterations. Array contraction can still be performed in such cases, but the minimum memory requirement must be calculated differently for the flow dependences. Moreover, the transformed code will be more complex.

We now compute the minimum memory requirement for the given loop nesting. First, we want to compute the minimum memory requirement to satisfy a single flow dependence with $w$ as the source and $r$ as the read reference. Notice that if there exists an extremely large shifting factor, then the coalesced distance may exceed $\Pi_{k=1}^{n} b_k$, which is the total iteration count of the innermost loop body. Suppose $w$ and $r$ are in the same loop nest before fusion and it is impossible for $w$ to take place before $r$ in the same iteration. In this case, no additional array element is written by $w$ before $r$ occurs in the same iteration of the fused loop, hence no need for the extra memory location. Under Assumption 1, the minimum memory requirement for this flow dependence equals either $\Pi_{k=1}^{n} b_k$ or the coalesced distance, whichever is smaller. In all other cases under Assumption 1, the minimum memory requirement for the flow dependence equals either $\Pi_{k=1}^{n} b_k$ or the coalesced distance plus one, whichever is smaller. In order to distinguish these two cases in the mathematical framework below, we designate an *extension vector* $\vec{o_e}$ for each flow dependence $e$ such that $\vec{o_e}$ equals $\vec{0}$ if $r$ precedes $w$ in the same loop body and $\vec{o_e}$ equals $(0, \ldots, 0, 1)$ otherwise.

Next, we want to compute the minimum memory size required to satisfy all flow dependences with $w$ as the source. This equals the maximum value of the required memory size for all flow dependences with $w$ as the source.

**Lemma 4** *For a reference $w$ in $L_i$ which writes to a local array region, under Assumption 1, the memory required to satisfy all flow dependences which have $w$ as the source equals*

$$\min(\max\{(\vec{p_j} + \vec{d_e} + \vec{o_e} - \vec{p_i})\vec{s}^T | \forall e \textit{ from } L_i \textit{ to } L_j \textit{ due to } w\}, \Pi_{k=1}^{n} b_k). \tag{4}$$

We call $(\vec{d_e} + \vec{o_e})$ the *extended dependence distance* for the flow dependence edge $e$.

Lastly, we determine the minimum memory size required for all write references to local array regions. To simplify the discussion, we make the following assumption, under which the memory required for all write references is equal to the summation of the memory required for each write reference.

**Assumption 2** *The local array regions written by different write references do not overlap. All local array regions remain live until the end of the $L_m$ loop nest (in Figure 2(a)).*

We want to formulate the memory requirement minimization problem as a network flow problem which can be solved in polynomial time [12]. However, the *min* operator in Formula (4) makes the minimization problem complex. To remove such a *min* operator, we make the following assumption,

**Assumption 3** *In $G$, the sum of the absolute values $|d_k|$ of all data dependence distances at loop level $k$ is smaller than the trip count $b_k$.*

Under Assumptions 1 to 3, the following theorem formalizes our problem of minimizing the overall memory requirement.

**Theorem 1** *Given the $m$ loop nests in Figure 2(a) and the loop dependence graph $G$, let $\tau_i$ be the number of write references to local array regions in each loop nest $L_i$. Let $e$ represent an arbitrary data dependence in $E$, say from $L_i$ to $L_j$, which may be a flow, anti-, or output dependence. Let $e_1$ represent an arbitrary flow dependence in $E$, say from $L_i$ to $L_j$, due to a local array region written by a reference $w_{i,k}(1 \leq k \leq \tau_i)$. Under loop shifting and loop fusion, the minimum memory requirement is determined by the following formula.*

$$\text{Minimize } \Sigma_{i=1}^{m} \Sigma_{k=1}^{\tau_i} \vec{M_{i,k}} \vec{s}^T \tag{5}$$

*subject to*

$$(\vec{p_j} + \vec{d_e} - \vec{p_i})\vec{s}^T \geq 0, \forall e \in E \text{ from } L_i \text{ to } L_j, \textit{and} \tag{6}$$

$$\vec{M_{i,k}} \vec{s}^T \geq (\vec{p_j} + \vec{d_{e_1}} + \vec{o_{e_1}} - \vec{p_i})\vec{s}^T, \forall e_1 \in E \text{ from } L_i \text{ to } L_j \text{ due to } w_{i,k}, 1 \leq k \leq \tau_i. \tag{7}$$

**Proof** See [1]. □

We call the problem defined in Theorem 1 as **Problem 0**. Given the shifting vectors, the expression $\vec{M_{i,k}} \vec{s}^T$ in constraint (7) defines the minimum memory requirement among all flow dependences originating from the same reference $w_{i,k}$ in $L_i$ which writes to a local array region. According to Lemma 3, constraint (6) guarantees legal loop fusion after shifting and coalescing. The objective function (5) defines the minimum memory size required for all write references. Thus, our array size minimization problem is to determine the shifting vectors which satisfy the constraints stated in the theorem above such that the total memory requirement is minimized.

According to the following lemma, loop coalescing is just an imagined intermediate transformation used to formulate **Problem 0**. After solving **Problem 0**, we do not need to actually perform it during code generation.

**Lemma 5** *For any set of shifting vectors which optimally solve* **Problem 0***, we can find a set of shifting vectors which allow legal fusion of the loop nests without coalescing and result in the same minimum memory requirement.*

**Proof** See [1]. □

Next, we reduce **Problem 0** to a minimum-cost network flow problem which can be solved optimally in polynomial time.

## 4 Reducing to a Network Flow Problem

Constraint (6) in Theorem 1 can be transformed to

$$\vec{p_i}\vec{s}^T - \vec{p_j}\vec{s}^T \leq \vec{d_e}\vec{s}^T, \forall e \in E \textit{ from } L_i \textit{ to } L_j. \tag{8}$$

Constraint (7) can be transformed to

$$\vec{p_j}\vec{s}^T - (\vec{p_i} + \vec{M_{i,k}})\vec{s}^T \leq -(\vec{d_{e_1}} + \vec{o_{e_1}})\vec{s}^T, \forall e_1 \in E \textit{ from } L_i \textit{ to } L_j \textit{ due to } w_{i,k}, 1 \leq k \leq \tau_i. \tag{9}$$

Formulae (8) and (9) bear a great similarity to the constraint in the dual problem of the general network flow problem (see Ch. 9.4 by Ahuja et al. [13] for details.)

$$\pi(i) - \pi(j) - \alpha_{ij} \leq c_{ij}, \tag{10}$$

where $c_{ij}$ are nonnegative constants, and $\alpha_{ij} \geq 0$, $\pi(i)$ and $\pi(j)$ are variables. This motivates us to develop a sequence of transformations to reduce **Problem 0** to a minimum-cost network flow problem.

First, we transform **Problem 0** to **Problem 1** in order to reduce the number of constraints. We then unify the constraints by changing constraint (7) to the same form as constraint (6). Moreover, we transform the underlying graph to a directed acyclic graph (DAG). The resulting problem is called **Problem 2**. Since a minimum-cost network flow problem must have a nonnegative cost [13], we transform **Problem 2** to **Problem 3** to satisfy such a requirement. Finally, we reduce **Problem 3** to **Problem 4** which can be solved using the existing minimum-cost flow algorithms [13]. We provide a method to use the optimal solution of **Problem 4** to compute an optimal solution of **Problem 3**, which then leads to an optimal solution of **Problem 0**.

## 4.1 Removing Redundant Constraints

In this subsection, we simplify **Problem 0** by removing redundant constraints which are due to superfluous dependence edges. We simplify the loop dependence graph $G$ to $G_0 = (V_0, E_0)$. The graph $G_0$ has the same set of nodes as $G$, but it may have fewer dependence edges. The dependence edges in $E_0$ can be divided into two categories, namely the *L-edge*s and the *M-edge*s. The L-edges are used to determine the legality of loop fusion. The M-edges will determine the minimum memory requirement. All M-edges correspond to flow dependences in $E$. An L-edge, however, may correspond to a flow, an anti- or an output dependence in $E$. It is possible for an edge in $E_0$ to be classified both as an L-edge and an M-edge at the same time. The simplification process is as follows.

- Suppose there exist multiple flow dependences between a pair of nodes in $V$, and suppose they are due to the same write reference. We keep only the one with the lexicographically maximum extended dependence distance as an M-edge in $E_0$. (In Figure 3(b), the flow dependence from $L_1$ to $L_3$ marked by the distance vector $(0,1)$ will be made an M-edge in $E_0$.) To see why this is valid, consider two edges in $E$, $e_1$ and $e_2$, both from node $L_i$ to node $L_j$ such that $(\vec{d_{e_1}} + \vec{o_{e_1}}) \succeq (\vec{d_{e_2}} + \vec{o_{e_2}})$. Note that $\vec{o_{e_1}}$ and $\vec{o_{e_2}}$ may be identical or differ by $(0,\ldots,0,1)$. According to Assumption 3, $abs(\vec{d_{e_1}} + \vec{o_{e_1}} - \vec{d_{e_2}} - \vec{o_{e_2}}) \leq abs(\vec{d_{e_1}}) + abs(\vec{d_{e_2}}) + (0,\ldots,0,1) < \vec{b} + (0,\ldots,0,1)$ holds. From Lemma 1, we have $(\vec{p_j} + \vec{d_{e_1}} + \vec{o_{e_1}} - \vec{p_i})\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_2}} + \vec{o_{e_2}} - \vec{p_i})\vec{s}^T$. If $\vec{M_{i,k}}\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_1}} + \vec{o_{e_1}} - \vec{p_i})\vec{s}^T$ is true, then $\vec{M_{i,k}}\vec{s}^T \geq (\vec{p_j} + \vec{d_{e_2}} + \vec{o_{e_2}} - \vec{p_i})\vec{s}^T$ must also be true.

- If there exist multiple dependences between any pair of distinct nodes in $E$, we keep only the one with the lexicographically minimum distance as an L-edge in $E_0$. (In Figure 3(b), the dependence from $L_1$ to $L_3$ marked by the distance vector $(0,0)$ is made an L-edge in $E_0$.) To see why this is valid, again consider two edges in $E$, $e_1$ and $e_2$, both from node $L_i$ to node $L_j$ $(i \neq j)$ such that $\vec{d_{e_1}} \succeq \vec{d_{e_2}}$. It is clear that if $e_2$ satisfies constraint (6) in Theorem 1, then so does $e_1$. We do not keep L-edges from a node to itself since such dependences are preserved automatically after loop fusion.

Figure 3(c) shows the loop dependence graph $G_0$ after simplification of Figure 3(b).

Based on the above analysis, we have the following **Problem 1** which is equivalent to **Problem 0** but with fewer constraints:

$$\text{Minimize } \Sigma_{i=1}^{m}\Sigma_{k=1}^{\tau_i}\vec{M_{i,k}}\vec{s}^T \tag{11}$$

subject to

$$\vec{p_j}\vec{s}^T + \vec{d_e}\vec{s}^T - \vec{p_i}\vec{s}^T \geq 0, \forall \text{ L-edge } e = \langle L_i, L_j \rangle \in E_0, \textit{ and} \tag{12}$$

$$\vec{M_{i,k}}\vec{s}^T \geq \vec{p_j}\vec{s}^T + (\vec{d_e} + \vec{o_e})\vec{s}^T - \vec{p_i}\vec{s}^T, \forall \text{ M-edge } e = \langle L_i, L_j \rangle \in E_0 \textit{ due to } w_{i,k}, 1 \leq k \leq \tau_i. \tag{13}$$

## 4.2 Creating a DAG

Given a loop dependence graph $G_0$, we generate another graph $G_1 = (V_1, E_1)$ as follows.

- For each node $L_i \in G_0$, we create its *corresponding* node $\tilde{L_i}$ in $G_1$. Furthermore, for each node $L_i \in G_0$ such that $L_i$ has an outgoing M-edge and for each write reference $w_{i,k}$ $(1 \leq k \leq \tau_i)$ in $L_i$, we create a new node $\tilde{L_i}^{(k)}$ in $G_1$. This new node is called the *sink* of $\tilde{L_i}$ due to $w_{i,k}$.

- We annotate each node in $G_1$ with a *supply/demand* value. For each node $L_i \in G_0$, if $L_i$ has an outgoing M-edge, let the supply/demand of its corresponding node $\tilde{L_i}$ be $h(\tilde{L_i}) = \tau_i$. For each write reference $w_{i,k}$ $(1 \leq k \leq \tau_i)$ in $L_i$, we let $h(\tilde{L_i}^{(k)}) = -1$ for the sink node. For each node $L_i \in G_0$ which does not have an outgoing M-edge, let $h(\tilde{L_i}) = 0$ for its corresponding node.

- Suppose an M-edge $e = \langle L_i, L_j \rangle$ in $G_0$ is due to the write reference $w_{i,k}$ and its distance vector is $\vec{d_e}$. We add an edge $\langle \tilde{L_j}, \tilde{L_i}^{(k)} \rangle$ to $G_1$ with the *cost* $\vec{c'}\vec{s}^T$, where $\vec{c'} = -(\vec{d_e} + \vec{o_e})$ is called the *cost vector* of the edge.

- Suppose $e = \langle L_i, L_j \rangle$ is an L-edge in $G_0$ and its distance vector is $\vec{d_e}$. We add an edge $\langle \tilde{L_i}, \tilde{L_j} \rangle$ to $G_1$ with the *cost* $\vec{c'}\vec{s}^T$, where $\vec{c'} = \vec{d_e}$ is called the cost vector of the edge.

For the graph $G_0$ in Figure 3(c), Figure 4(a) shows the transformed graph $G_1$.

**Lemma 6** *The graph $G_1 = (V_1, E_1)$ is a simple DAG.*

**Proof** After making the simplification described in Section 4.1, $G_0$ has such a property that, between any pair of nodes, there exist at most one L-edge and at most one M-edge for each write reference in the source node. The graph transformation from $G_0$ to $G_1$ discussed above introduces additional nodes. As a result, self cycles are removed and there exist at most one edge between any pair of nodes in $G_1$. $\Box$

After constructing $G_1$, we define a variable vector $\vec{q}$ for each node $v_j \in V_1$ as follows.

- If $v_j$ is a "corresponding" node $\tilde{L_i}$ in $G_1$, we define

$$\vec{q_j} = \vec{p_i}. \tag{14}$$

- If $v_j$ is a sink $\tilde{L_i}^{(k)}$, we define

$$\vec{q_j} = \vec{M_{i,k}} + \vec{p_i}. \tag{15}$$

We define a new system, called **Problem 2**, over $\vec{q}$, using the cost $\vec{c_{ij}}\vec{s}^T$ introduced above for each edge $e = \langle v_i, v_j \rangle \in E_1$:

$$\text{Maximize } \Sigma_{i=1}^{|V_1|} h(v_i)\vec{q_i}\vec{s}^T \tag{16}$$

subject to

$$\vec{q_j}\vec{s}^T - \vec{q_i}\vec{s}^T + \vec{c_{ij}}\vec{s}^T \geq 0, \forall e = \langle v_i, v_j \rangle \in E_1. \tag{17}$$

**Theorem 2** **Problem 2** *is equivalent to* **Problem 1**.

**Proof** See Appendix B. $\Box$

## 4.3 Making the Cost Nonnegative

The cost $\vec{c'_{ij}}\vec{s}^T$ of the transformed graph $G_1$ may have a negative value. In order to derive a network flow problem (*c.f.* Section 4.4), we transform $G_1$ to $G_2$ such that the cost for all edges becomes nonnegative.

First, we determine the *lexical height vector* $\vec{z_j}$ for each node $v_j \in G_1$, using the algorithm in Figure 5, where the operator *lex_max* stands for the lexicographical maximum.

**Lemma 7** $\forall \langle v_i, v_j \rangle \in E_1, (\vec{z_j} - \vec{z_i} + \vec{c'_{ij}})\vec{s}^T \geq 0$.

**Proof** See Appendix C. □

We rename a variable vector

$$\vec{y_i} = \vec{q_i} - \vec{z_i}. \tag{18}$$

Clearly $\vec{q_i}\vec{s}^T = \vec{y_i}\vec{s}^T + \vec{z_i}\vec{s}^T$. The constraint (17) becomes

$$\vec{q_j}\vec{s}^T - \vec{q_i}\vec{s}^T + \vec{c'_{ij}}\vec{s}^T = \vec{y_j}\vec{s}^T - \vec{y_i}\vec{s}^T + (\vec{z_j} - \vec{z_i} + \vec{c'_{ij}})\vec{s}^T \geq 0, \forall e = \langle v_i, v_j \rangle \in E_1. \tag{19}$$

The objective function (16) becomes

$$\Sigma_{i=1}^{|V_1|} h(v_i)\vec{q_i}\vec{s}^T = \Sigma_{i=1}^{|V_1|} h(v_i)\vec{y_i}\vec{s}^T + \Sigma_{i=1}^{|V_1|} h(v_i)\vec{z_i}\vec{s}^T. \tag{20}$$

For each edge $\langle v_i, v_j \rangle \in E_1$, we define a new cost which equals $\vec{c_{ij}}\vec{s}^T$, where $\vec{c_{ij}} = \vec{z_j} - \vec{z_i} + \vec{c'_{ij}}$. This cost is nonnegative for all edges. After this new cost assignment, we rename the graph to $G_2 = (V_2, E_2)$ and define the following system (called **Problem 3**) over $G_2$:

$$\text{Maximize } \Sigma_{i=1}^{|V_2|} h(v_i)\vec{y_i}\vec{s}^T \tag{21}$$

subject to

$$\vec{y_j}\vec{s}^T - \vec{y_i}\vec{s}^T + \vec{c_{ij}}\vec{s}^T \geq 0, \forall e = \langle v_i, v_j \rangle \in E_2. \tag{22}$$

The equivalence between **Problem 3** and **Problem 2** is obvious from the discussions above.

In our example in Figure 4(a), the graph $G_1$ is transformed to graph $G_2$ shown in Figure 4(b), where $\vec{z}(\tilde{L_1}) = \vec{z}(\tilde{L_2}) = (0,0)$, $\vec{z}(\tilde{L_3}) = \vec{z}(\tilde{L_4}) = (1,0)$, $\vec{z}(\tilde{L_1}^{(1)}) = (1,2)$ and $\vec{z}(\tilde{L_2}^{(1)}) = (1,1)$. Note that $\vec{z}(v_i)$ is a more explicit notation for $\vec{z_i}$. Likewise, $\vec{p}(v_i)$ (or $\vec{q}(v_i)$) is a more explicit notation for $\vec{p_i}$ (or $\vec{q_i}$).

## 4.4 A Network Flow Problem

We now reduce **Problem 3** to a network flow problem, **Problem 4**, which can be solved by existing algorithms. We then show the equivalence between **Problem 3** and **Problem 4**. **Problem 4** is defined by (23)-(25) below:

$$\text{Minimize } \Sigma_{e=\langle v_i,v_j\rangle \in E_2}(f(e)\vec{c_{ij}}\vec{s}^T) \tag{23}$$

subject to

$$\Sigma_{e=\langle v_i,.\rangle \in E_2} f(e) - \Sigma_{e=\langle .,v_i\rangle \in E_2} f(e) = h(v_i), 1 \leq i \leq |V_2|, and \tag{24}$$

$$f(e) \geq 0, \forall e \in E_2. \tag{25}$$

In order to use the existing minimum-cost flow algorithms, the following requirements must be met [13]:

1. All data (the cost $\vec{c_{ij}}\vec{s}^T$, the supply/demand value $h(v_i)$ and the flow $f(e)$) are integral.

2. The graph $G_2$ is directed.

3. The supply/demand values satisfy the condition $\Sigma_{i=1}^{|V_2|} h(v_i) = 0$ and **Problem 4** has a feasible solution.

4. All costs $\vec{c_{ij}}\vec{s}^T$ are non-negative.

**Problem 4** obviously satisfies requirements (2) and (4). Both the supply/demand $h(v_i)$ and the flow $f(e)$ are integer numbers, which meet requirement (1). However, the cost is represented as an inner product of $\vec{c_{ij}}$ and $\vec{s}$, which may contain symbolic terms if $\vec{s}$ has symbolic terms. Later in this subsection, we shall discuss how to deal with symbolic terms in $\vec{s}$ when solving **Problem 4**. From the construction of $G_1$ and $G_2$, we see that $\Sigma_{i=1}^{|V_2|} h(v_i) = 0$. **Problem 4** meets requirement (3) if we can show that it has a feasible solution.

We construct a feasible solution for **Problem 4** as follows. Recall that $G_1$ and $G_2$ have the identical nodes, edges and supply/demand values. For each node $L_i \in G_0$ and for each write reference $w_{i,k}$ $(1 \leq k \leq \tau_i)$ in loop $L_i$, suppose $L_i$ has one or more outgoing M-edges due to $w_{i,k}$. We arbitrarily pick one, say $\langle L_i, L_j \rangle$. Let $\tilde{L_j}$ be the corresponding node of $L_j$ in $G_2$ and let $\tilde{L_i^{(k)}}$ be the sink of $L_i$ in $G_2$ due to $w_{i,k}$. When we assign the flow values in $G_2$, we let $f(\langle \tilde{L_i}, \tilde{L_j} \rangle) = 1$ and $f(\langle \tilde{L_j}, \tilde{L_i}^{(k)} \rangle) = 1$. For the remaining edges in $G_2$ which do not get flow values as described above, we let their flow values $f(e)$ to be 0. Clearly, such a flow assignment satisfies constraints (24) and (25). Hence, it is a feasible solution for **Problem 4**.

**Problem 4** can be solved by existing algorithms with pseudo-polynomial or polynomial time complexity [13]. These include the *successive shortest path algorithm*, the *double scaling algorithm* and the *enhanced capacity scaling algorithm*, among others. As an example, Figure 6 shows the enhanced capacity scaling algorithm (Section 10.7 by Ahuja et al. [13], page 389). The main idea of this algorithm is to identify and augment along edges with *sufficient large* flow values, in order to reduce the number of flow augmentation in *residual networks*. According to Ahuja et al. [13], the outer **while** loop iterates $O(|V_2| log |V_2|)$ times. It takes time $O(|E_2|)$ to update abundant components and reinstate the imbalance property in each outer **while** iteration. The total number of iterations for the inner **while**

loop is $O(|V_2|^2 log|V_2|)$. The overall complexity for this algorithm is $O((|E_2|log|V_2|)(|E_2| + |V_2|log|V_2|))$.

The enhanced capacity scaling algorithm has two steps which involve the costs $\vec{c_{ij}}\vec{s}^T$. One of the steps computes the shortest path using the reduced costs, $\vec{c_{ij}}\vec{s}^T - \vec{\pi_i}\vec{s}^T + \vec{\pi_j}\vec{s}^T$, as the edge lengths. The other step computes $\pi_i$ which is also represented as the inner product of a certain vector and $\vec{s}$. In these two steps, in case $\vec{s}$ contains symbolic terms, the normal computation can still be applied, by using $\vec{u_1}\vec{s}^T + \vec{u_2}\vec{s}^T = (\vec{u_1} + \vec{u_2})\vec{s}^T$, for example. If we want to compare $\vec{u_1}\vec{s}^T$ with $\vec{u_2}\vec{s}^T$, we can simply compare $\vec{u_1}$ with $\vec{u_2}$, but at the same time we generate a condition $abs(\vec{u_1} - \vec{u_2}) < \vec{b}$. This condition, according to Lemma 1, will guarantee $\vec{u_1}\vec{s}^T > \vec{u_2}\vec{s}^T \Leftrightarrow \vec{u_1} \succ \vec{u_2}$. The compiler can generate two versions of code, one for the original loop nest and the other for applying array contraction. The version that applies array contraction is executed at run time if all the conditions generated such (for cost comparison) hold. Such a treatment for comparison may potentially miss more memory minimization opportunities than a more precise comparison method. Nonetheless, we expect our methods to cover most cases in practice, since the dependence distances are usually much smaller than the loop iteration counts.

In this paper, we do not explore the details of the enhanced capacity scaling algorithm or any other network algorithms. The readers are referred to Ahuja et al. [13] for further details. Next, we show that **Problem 3** is equivalent to **Problem 4**.

### 4.4.1 Solving Problem 3 via Problem 4

First, we show that the objective value of **Problem 3** is no greater than that of **Problem 4**. This is because, after multiplying both sides of equation (24) by $\vec{y_i}\vec{s}^T$ and adding all $|V_2|$ equations together, we have

$$\begin{aligned}&\Sigma_{i=1}^{|V_2|}h(v_i)\vec{y_i}\vec{s}^T\\&=\Sigma_{i=1}^{|V_2}(\Sigma_{e=\langle v_i,.\rangle\in E_2}f(e)-\Sigma_{e=\langle .,v_i\rangle\in E_2}f(e))\vec{y_i}\vec{s}^T\\&=\Sigma_{e=\langle v_i,v_j\rangle\in E_2}f(e)(\vec{y_i}\vec{s}^T-\vec{y_j}\vec{s}^T).\end{aligned}$$

Combining this with formulas (22) and (25), we have

$$\Sigma_{i=1}^{|V_2|}h(v_i)\vec{y_i}\vec{s}^T \le \Sigma_{e=\langle v_i,v_j\rangle\in E_2}f(e)\vec{c_{ij}}\vec{s}. \tag{26}$$

Next, we show that given an optimal solution $f^*(e)$ for **Problem 4** we can find proper values of $\vec{y_i^*}$ which satisfy constraint (22) and the equality $\Sigma_{i=1}^{|V_2|}h(v_i)\vec{y_i^*}\vec{s}^T=\Sigma_{e=\langle v_i,v_j\rangle\in E_2}f^*(e)\vec{c_{ij}}\vec{s}$. Thus, $\vec{y_i^*}$ will be optimal for **Problem 3**. To show this, given any optimal solution $f^*(e)$ of **Problem 4**, we take the following steps to build the residual network [13], denoted by $G_2'$, for the network $G_2$:

- For each node $v_i$ in $G_2$, we also make it a node $v_i$ in $G_2'$.
- For each edge $e=\langle v_i,v_j\rangle\in E_2$, we also make it an edge in $G_2'$ and mark it by the *distance* $\vec{c_{ij}}\vec{s}^T$.
- If $f^*(e)>0, e=\langle v_i,v_j\rangle\in E_2$, we add an edge $\langle v_j,v_i\rangle$ and mark it by the *distance* $-\vec{c_{ij}}\vec{s}^T$ in $G_2'$.
- We add a new node $S$ to $G_2'$ with the supply/demand value 0. For each node $v_i$ in $G_2$ which does not have predecessors, we add an edge $\langle S,v_i\rangle$ to $G_2'$ with the distance 0

and with the flow value 0. (The node $S$ is added to $G_2$ to facilitate the shortest path algorithm in the work below, which does not affect the optimality of **Problem 3**.)

In order to obtain an optimal solution for **Problem 3**, we want to calculate the shortest path from $S$ to each of the remaining nodes in $G_2'$. This requires the graph to contain no cycle that has a negative total distance. The following lemma shows that $G_2'$ satisfies such a requirement.

**Lemma 8** *The residual network $G_2'$ has no cycles with negative total distance.*

**Proof** See Appendix D. □

We calculate the shortest path from $S$ to each of the remaining nodes in $G_2'$. Suppose we have the shortest distance value $\vec{x_i}\vec{s}^T$ for each node $v_i$ in $G_2'$ except $S$. Whenever necessary, we formulate conditions similar to those in requirement (1) of **Problem 4**, which will be checked at run time in order to obtain valid results. For example, the following inequality must hold for every edge $\langle v_i, v_j \rangle$,

$$abs(\vec{x_j} - (\vec{x_i} + \vec{c_{ij}})) < \vec{b}. \tag{27}$$

If this inequality cannot be verified for any edge before run time, then we must insert it in the program as a condition to be checked at run time. If the condition is not met, then the program must not execute the version of code which has the arrays contracted. Next, we prove that

$$\vec{y_i^*} = -\vec{x_i} \tag{28}$$

is a solution to **Problem 3** and the objective value of **Problem 3** equals that of **Problem 4**. To prove this, we first observe that the following constraint

$$(-\vec{x_j}\vec{s}^T) - (-\vec{x_i}\vec{s}^T) + \vec{c_{ij}}\vec{s}^T \geq 0 \tag{29}$$

is clearly satisfied because of the shortest distance property for the edge $e = \langle v_i, v_j \rangle$. Hence, $-\vec{x_j}$ is a feasible solution to **Problem 3**. Second, if $f^*(e) > 0$, then the inequality

$$(-\vec{x_i}\vec{s}^T) - (-\vec{x_j}\vec{s}^T) + (-\vec{c_{ij}}\vec{s}^T) \geq 0 \tag{30}$$

holds because of the shortest distance property for the edge $\langle v_j, v_i \rangle$. Combining inequalities (29) and (30), the following equality (31) must hold.

$$(-\vec{x_j}\vec{s}^T) - (-\vec{x_i}\vec{s}^T) = -\vec{c_{ij}}\vec{s}^T \ \ \textit{if}\ f^*(e) > 0. \tag{31}$$

With the above facts established, we get

$$\begin{aligned}
&\Sigma_{i=1}^{|V_2|} h(v_i)\vec{y_i^*}\vec{s}^T \\
&= \Sigma_{e=\langle v_i,v_j\rangle \in E_2} f^*(e)(\vec{y_i^*}\vec{s}^T - \vec{y_j^*}\vec{s}^T) \\
&= \Sigma_{e=\langle v_i,v_j\rangle \in E_2} f^*(e)((-\vec{x_i}\vec{s}^T) - (-\vec{x_j}\vec{s}^T)) \\
&= \Sigma_{e=\langle v_i,v_j\rangle \in E_2} f^*(e)\vec{c_{ij}}\vec{s}^T,
\end{aligned}$$

which proves the equivalence between **Problem 3** and **Problem 4**.

## 4.5 Solving Problem 0

Figure 7 lists the algorithm to compute a set of optimal shifting vectors for array contraction, namely for **Problem 0**. We illustrate this algorithm through Example 2 in Figure 3, where $\vec{s} = (\mathrm{JN} - 1, 1)$.

1. Assumptions 1 and 2 hold for local array regions $ZA(2 : JN, 2 : KN)$ and $ZB(2 : JN, 2 : KN)$. In order to satisfy Assumption 3, the condition $(3, 4) < \vec{b}$ must be checked at run time.

2. By utilizing a network flow algorithm described by Ahuja et al. [13], we get an optimal solution for **Problem 4** defined over the graph in Figure 4(b):

   $f(\langle \tilde{L_1}, \tilde{L_3} \rangle) = f(\langle \tilde{L_2}, \tilde{L_4} \rangle) = f(\langle \tilde{L_3}, \tilde{L_1^{(1)}} \rangle) = f(\langle \tilde{L_4}, \tilde{L_2^{(1)}} \rangle) = 1$, the rest of the flow values being 0. The optimal objective function value is $(1,0)\vec{s}^T$. Note that the condition $(1,0) < \vec{b}$ must hold in order for this result to be valid.

3. Utilizing the equivalence between **Problem 3** and **Problem 4**, we compute the optimal solution for **Problem 3** in Figure 4(b). We have $\vec{y_i} = (1,0)$ for $\tilde{L_1}$ and $\vec{y_i} = (0,0)$ for all the other nodes. The optimal objective function is $(1,0)\vec{s}^T$. The condition $(1,0) < \vec{b}$ must hold in order for this result to be valid.

4. We compute the optimal solution for **Problem 2** in Figure 4(a) based on equations (18) and (20). We get $\vec{q}(\tilde{L_1}) = (1,0)$, $\vec{q}(\tilde{L_2}) = (0,0)$, $\vec{q}(\tilde{L_3}) = \vec{q}(\tilde{L_4}) = (1,0)$, $\vec{q}(\tilde{L_1}^{(1)}) = (1,2)$ and $\vec{q}(\tilde{L_2}^{(1)}) = (1,1)$. The optimal value for the objective function (16) is $(3,3)\vec{s}^T$.

5. We compute the optimal solution for **Problem 1** (and hence **Problem 0**) based on formulas (14) and (15). We have $\vec{p}(L_1) = (1,0)$, $\vec{p}(L_2) = (0,0)$, $\vec{p}(L_3) = \vec{p}(L_4) = (1,0)$, $\vec{M_{1,1}} = (0,2)$ and $\vec{M_{2,1}} = (1,1)$.

6. The conjunction of all the conditions in steps 1-3 can be written as $(3,4) < \vec{b}$. This condition will be checked at run time. If it is true, the transformed code will be executed. Otherwise, the original code is executed.

Given $G = (V, E)$, it takes $O(n|E|)$ time to simplify **Problem 0** to **Problem 1**. For the transformed LDG $G_1 = (V_1, E_1)$, both $|V_1| \leq |V| + |E|$ and $|E_1| \leq |E|$ hold. It takes

$O(n|E_1|)$ time to transform $G_0$ to $G_1$ and to transform $G_1$ to $G_2$. As mentioned before, polynomial-time algorithms exist for solving **Problem 3** and **Problem 4** [13].

## 5 Code Generation

In this section, we discuss several issues concerning code generation based on the solution computed by the scheme we presented in Section 4.5 for **Problem 0**.

First of all, we need to determine an optimal solution which does not require loop coalescing. As stated in Lemma 5, we can always find such an optimal solution (using the method described in its proof). We then apply loop shifting and loop fusion based on this optimal set of shifting factors.

In the given collection of loops, it is possible for a reference $r$ to access both local regions and non-local regions of any array. Only the local array regions will be contracted. We must create two copies of $r$, one to access the non-local array regions and the other to the local array regions. Correspondingly, we generate two branches in an IF statement such that the local array regions and the non-local array regions are referenced in different branches. The IF condition verifies whether the current loop index values make $r$ refer to the local or the non-local regions.

An array may consist of both local array regions and non-local array regions. If we contract one or more local array regions of an array, we separate these regions from the rest by creating new arrays, some for the contractable local regions and the rest for the remaining array regions. For each contractable local array region $R$ written by a reference $w_{i,k}$, we take the value of $\vec{M_{i,k}}\vec{s}^T$ from the optimal solution for **Problem 0** defined by constraints (5)-(7).

We create a one-dimensional array $R'$ with a size of $\vec{M_{i,k}}\vec{s}^T$. The array regions which are not contracted are also renamed to additional new arrays. The original array is removed.

There exist various ways to declare the lower bound of the new array $R'$ for each contracted local array region and to generate the new array subscripts. We do not cover all details in this paper, since the exact treatment depends on the subscript patterns. Instead, we discuss a common case in which the write reference writes adjacent memory locations in consecutive loop iterations under the unit stride. For such a common case, we let $R'$ have the lower bound of 1. We then replace each reference to $R$ by a reference to $R'$. The reference to $R'$ has a new array subscript which equals the linearized expression of the old subscript for the local array region $R$, modulo the size of $R'$, plus 1. (For certain array subscript patterns of $R$, it may be possible to adjust the lower bound of $R'$, and hence the array subscripts, to make the address computation simpler. Furthermore, if $\vec{M_{i,k}}\vec{s}^T$ is a small known constant, we can create several scalar variables instead of a single array, as in Example 1 in Figure 1. We also omit further details.)

To illustrate the code generation, take Example 2 in Figure 3. Loop shifting is applied to the loop nests using the optimal shifting factors computed in Section 4.5. The loop nests are then fused. Next, we split the arrays into local and non-local array regions. Two new arrays *ZA0* and *ZB0* are created for the non-local array regions $ZA(1, 2 : KN)$ and $ZB(2 : JN, KN + 1)$, respectively. We use $ZA'(2 : JN, 2 : KN)$ and $ZB'(2 : JN, 2 : KN)$ to represent the contractable local array regions for $ZA(2 : JN, 2 : KN)$ and $ZB(2 : JN, 2 : KN)$ respectively. The local array region $ZA'$ is then contracted to an array *ZA1* of size two. For the local array region $ZB'$, we create a new contracted array $ZB1(1 : JN)$. The references to the old local array regions are replaced by references to the new local array regions, as shown in the transformed code in Figure 8. Notice that the statements in the original loops L3 and L4 have two versions in the new code, because array references to $ZA$ and $ZB$ refer to both local and non-local array regions.

## 6 Experimental Results

We have implemented our memory reduction technique in a research compiler, Panorama [8]. Panorama is a source-to-source translator for Fortran F77 programs, which implements automatic parallelization in addition to a number of loop transformation techniques, including loop fusion, loop shifting, loop tiling, and so on. We implemented a network flow algorithm, *enhanced capacity scaling algorithm* [13], to solve **Problem 4**. To measure its effectiveness, we applied our memory reduction technique to 20 benchmark programs. We measured the program execution speed on a SUN UltraSPARC II uniprocessor workstation and on a MIPS R10K processor within an SGI Origin 2000 multiprocessor. In this paper, we show the memory reduction rate and the performance results on the UltraSPARC II. More experimentation details, including the R10K results, can be found in related papers [1, 12].

Table I 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 known compiler algorithms [1] to fit. 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 the different versions of the program `swim` from SPEC95 and SPEC2000, we denote the SPEC95 version by `swim95` and the SPEC2000 version by `swim00`. Program `swim00` is almost identical to `swim95` except for its larger data size. For `combustion`, we increase the array size (N1 and N2) from 1 to 10 to make the execution last for at least several seconds. Programs `climate`, `laplace-jb`, `laplace-gs` and all the Purdue set problems are from an HPF benchmark suite maintained by Rice University [14, 15]. All these benchmarks are written in F77 except `lucas` which is in F90. We manually apply our technique to `lucas`. Among these 20 benchmark programs, our algorithm finds that loop shifting is not needed to fuse the target loops in the purdue-set programs, `lucas`, `LL14`, `climate` and `combustion`. For each of the benchmarks in Table I, 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.

To produce the machine codes, we use the native compiler, Sun WorkShop 6 update 1, to compile both original programs and transformed programs compiled by Panorama at the source level. For the original `tomcatv` code, we use the compiler flag "-fast -xchip=ultra2 -xarch=v8plusa -xpad=local:23". For all versions of `swim95` and `swim00`, we use the flag "-fast -xchip=ultra2 -xarch=v8plusa -xpad=common:15". For all versions of `combustion`, we simply use "-fast" because it produces better-performing codes than using other flags. For all other codes, we use the flag "-fast -xchip=ultra2 -xarch=v8plusa -fsimple=2".

Figure 9 compares the code sizes and the data sizes of the original and the transformed codes. Each benchmark program has two vertical bars, the left for the original programs and the right for the programs transformed by our technique. We compute the data size by adding the size of global data in common blocks and the size of local data defined in the main program and other subroutines/functions invoked at runtime. The data size of 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. The data size of each transformed program, and the code sizes for both original and transformed programs, are computed as a percentage to the data size of the corresponding original program. 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 49% for all benchmarks. For several small `purdue` benchmarks, the reduction rate is almost 100%. As Figure 9 shows, the code sizes remain almost the same before and after transformation for the tested 20 benchmarks.

Figure 10 shows the normalized execution time for the 20 test programs. On average, our technique achieves a speedup of 1.73 over the original code. Among all the benchmarks listed above, only `combustion, purdue-07 and purdue-08` fit the program model in previous work [2]. In those cases, the previous algorithm [2] will derive the same result as ours. For the rest of the cases, the previous algorithm [2] does not apply. Therefore, there is no need to list those results.

We also measured the impact of our technique on compile time. It takes Panorama less than 2 seconds to run all its passes on each of the test programs. Our technique, including loop dependence graph building and code generation, often takes one-tenth to one-fifth of the total time consumed by Panorama. If loop shifting is not needed for loop fusion, then the native compiler compiles the transformed source codes faster than the original program. This is because the transformed program contains fewer loops. On the other hand, if loop shifting is applied, then another transformation called *loop peeling* often follows after the loops are fused. Loop peeling strips off a few loop iterations, forming new loop nests sometimes. With loop peeling, IF statements might be removed from the fused loop body, making the main loop nests better for optimization and potentially run faster. The increased number of loop nests, however, may increase the time consumed by the native compiler. For example, on the UltraSPARC II, `tomcatv` compile time is increased by 50%. The maximum compile time increase is for `swim95` with 200% where procedure inlining also contributes certain compile time increase.

Several transformations, such as loop interchange, circular loop skewing [5], loop distribution, etc., are performed during source-to-source translation to make loop nests satisfy our assumptions [1]. In Panorama, we keep a copy of the original loop nesting while the transformations are applied. If none of the transformations succeed in making the loop nesting satisfy our assumptions, Panorama restores the original loop nesting. Nonetheless, it is useful to evaluate the potential performance impact of such transformations, assuming that we do not restore the original loop nesting. For example, loop interchange is performed on `tomcatv` to increase opportunities for loop fusion. Such a loop interchange, however, will make spatial locality suffer in `tomcatv` because the array accesses are no longer in the unit stride. Our experiment performed on the UltraSPARC II shows that, if we do not follow the loop interchange by array contraction, the code runs five times as slow as the original code. On the other hand, circular loop skewing alone does not seem to have a direct impact on the performance. This loop transformation is performed on both `swim95` and `swim00` to remove several dependences between adjacent loop nests which have long backward dependence distances. If we do not follow circular loop skewing by array contraction, the performance is almost the same as the original program. Loop distribution performed at the source level also seems to have insignificant performance impact. This is because most production compilers for high performance computers are able to fuse those loops which are created by loop distribution. For example, for `LL18`, if we disable the loop fusion feature of the native compiler on the UltraSPARC II, the program with loops distributed will perform worse by five percent than the original code. However, if we do not disable the loop fusion feature, the program with loops distributed performs the same as the original code. The native compiler fuses the loops back before generating the machine code.

It is possible for our array contraction technique to bring no benefit or even a slowdown to the program. For example, as Figure 10 shows, our technique has negligible performance impact on `lucas`. This is because array contraction is applicable to only a subset of the loops in the program. This subset consumes a small portion of the total execution time. The execution time of both `purdue-13` and `laplace-gs` is increased by as little as 2%. In `purdue-13`, there are two loops before our transformation, one with just one basic block and the other with a loop-variant condition inside the loop body. The first loop can be software pipelined and the second cannot. The fused loop, which contains a loop-variant condition inside the loop body, cannot be software pipelined. For the same reason, `laplace-gs` gets an insignificant amount of performance penalty by array contraction.

## 7 Related Work

Among the related previous work, the one closest to ours is conducted by Fraboulet *et al* [2]. They 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. However, this optimal algorithm does not apply to a collection of multi-level loop nests. It assumes a single-level loop whose data dependences have constant distances. For a tight loop nest of multiple levels, they present a heuristic which handles one level at a time, which in general may not be optimal. We should also mention that their memory requirement model does not apply when the dependence distances exceed certain values or when the loop body contains guards as the result of loop shifting before loop fusion. In another contribution, Fraboulet *et al.* present an integer programming algorithm to minimize memory space for a collection of loops [3]. Their algorithm, however, does not perform loop shifting as illustrated in Figure 1(b), and hence is unable to minimize memory requirement where shifting is required. We have presented in this paper a method to overcome these difficulties by combining loop shifting, loop fusion and array contraction. We set up a network flow problem which, through an elaborative series of steps, models the effect of such a combination.

Loop fusion has been studied extensively in various contexts which are different from ours. To name a few, Kennedy and McKinley prove that maximizing data locality by loop fusion is NP-hard [17]. Singhai and McKinley present *parameterized loop fusion* to improve parallelism and cache locality simultaneously [18]. Manjikian and Abdelrahman present a *shift-and-peel* technique to increase opportunities for loop fusion [19].

Gao *et al.* combine loop fusion and array scalarization to improve register utilization [20]. They do not consider contracting to low-dimensional arrays or using loop shifting. Lim *et al.* combine loop blocking, loop fusion and array scalarization to exploit parallelism [21]. They do not apply loop shifting during their optimization. Cociorva *et al.* use loop tiling to help balance the storage usage and the amount of computation [22]. Their method does not apply loop shifting before loop fusion. In our work, we do not analyze how array contraction affects the amount of computation. Instead, we let the later optimization passes, such as invariant code motion, to further improve the transformed code to reduce computation. Pike and Hilfinger present a framework for loop tiling and array contraction [23]. Their technique can also apply loop shifting before loop fusion. They conduct extensive trial runs of the program which enumerate a large number of parameters for loop transformations. They then select the parameter values which result in the best performance. When the range of parameter values is wide, the enumeration approach may be too tedious and time consuming to conduct. In such cases, we believe that our technique is more suitable, since it determines the transformation parameters at compile time, depending primarily on static program analysis.

Compared to our previous work [12], the optimization problem is formulated in a substantially better way in this paper. We improve our previous discussion on the network flow solution by presenting a complete solution which includes several intermediate steps and by explicitly providing a method to solve the original optimization problem.

## 8 Conclusion

In this paper, we have presented a technique to reduce the virtual-memory space required to execute a collection of multi-level loop nests. Our technique combines loop shifting, loop fusion and array contraction. We reduce the memory reduction problem to a network flow problem which can be solved optimally in polynomial time. A code generation guideline is also presented. We conduct experiments on 20 benchmarks. Our experimental results show that our memory reduction technique reduces memory consumption by 49% on average, counting both code and data sizes. The code size remains almost the same as before the transformation.

## Acknowledgements

This work is sponsored in part by National Science Foundation through grants CCR-950254, CCR-9975309, and ITR/ACR-0082834, and by a gift from Sun Microsystems. The authors thank Rong Xu for his contribution to the implementation of the scheme described in this work. The authors also thank the reviewers for their careful reviews and useful suggestions.

## Appendix A: Proof of Lemma 1

In the first part of the lemma, we have $abs(\vec{u}) < \vec{b}$. If $\vec{u} = \vec{0}$, we immediately have $\vec{u}\vec{s}^T = 0$. Suppose $\vec{u} \succ \vec{0}$. We prove $\vec{u}\vec{s}^T > 0$ as follows. Since $abs(\vec{u}) < \vec{b}$, we have $\vec{u} \geq \vec{v} = (0, \ldots, 0, 1, 1 - b_{k+1}, \ldots, 1 - b_n)$, $1 \leq k \leq n$, assuming the first non-zero element in $\vec{u}$ is the $k$th element. Since $\vec{u} - \vec{v} \geq \vec{0}$, we have $\vec{u}\vec{s}^T - \vec{v}\vec{s}^T = (\vec{u} - \vec{v})\vec{s}^T \geq 0$, hence $\vec{u}\vec{s}^T \geq \vec{v}\vec{s}^T$. Since $\vec{v}\vec{s}^T = s_n > 0$, we have $\vec{u}\vec{s}^T > 0$.

Similarly, we can prove that $\vec{u} \prec \vec{0} \Rightarrow \vec{u}\vec{s}^T < 0$. Therefore, if $\vec{u}\vec{s}^T > 0$, then $\vec{u} \succ \vec{0}$ must be true. The first part of the lemma is proved.

Next, we prove the second part of the lemma, in which we have $abs(\vec{u}) < \vec{b} + (0, \ldots, 0, 1)$. If $\vec{u} = \vec{0}$, we immediately have $\vec{u}\vec{s}^T = 0$.

Suppose $\vec{u} \succ \vec{0}$. We prove $\vec{u}\vec{s}^T \geq 0$ as follows. Since $abs(\vec{u}) < \vec{b} + (0, \ldots, 0, 1)$, we have $\vec{u} \geq \vec{v} = (0, \ldots, 0, 1, 1 - b_{k+1}, \ldots, -b_n)$, $1 \leq k \leq n$, assuming the first non-zero element in $\vec{u}$ is the $k$th element. Since $\vec{u} - \vec{v} \geq \vec{0}$, we have $\vec{u}\vec{s}^T - \vec{v}\vec{s}^T = (\vec{u} - \vec{v})\vec{s}^T \geq 0$, hence $\vec{u}\vec{s}^T \geq \vec{v}\vec{s}^T$. Since $\vec{v}\vec{s}^T = 0$, we have $\vec{u}\vec{s}^T \geq 0$.

## Appendix B: Proof of Theorem 2

We have

$\Sigma_{j=1}^{|V_1|} h(v_j)\vec{q_j}\vec{s}^T$

$= \Sigma_{\tau_i>0}(\tau_i\vec{q_i}\vec{s}^T - \Sigma_{k=1}^{\tau_i}((\vec{M_{i,k}} + \vec{q_i})\vec{s}^T)) + \Sigma_{\tau_i=0}0$

$= -\Sigma_{i=1}^{m}\Sigma_{k=1}^{\tau_i}\vec{M_{i,k}}\vec{s}^T.$

Hence the maximum of the objective function (16) is equivalent to the minimum of the objective function (11). For each edge $e_1 = \langle \tilde{L_i}, \tilde{L_j} \rangle$ in $E_1$, the inequality (17) is equivalent to

$$\vec{p_j}\vec{s}^T - \vec{p_i}\vec{s}^T + \vec{d_e}\vec{s}^T \geq 0, \tag{32}$$

where $e$ is an L-edge in $E_0$ from $L_i$ to $L_j$. Inequality (32) is equivalent to (12). For each edge $e_1 = \langle \tilde{L_j}, \tilde{L_i}^{(k)} \rangle$ in $E_1$, the inequality (17) is equivalent to

$$\vec{M_{i,k}}\vec{s}^T + \vec{p_i}\vec{s}^T - \vec{p_j}\vec{s}^T - (\vec{d_e} + \vec{o_e})\vec{s}^T \geq 0, \tag{33}$$

where $e$ is an M-edge in $E_0$ from $L_i$ to $L_j$ due to the write reference $w_{i,k}$. Inequality (33) is equivalent to (13).

## Appendix C: Proof of Lemma 7

Suppose $\vec{z_j} = \vec{z_t} - \vec{c_{tj}}$ where $\langle v_t, v_j \rangle \in E_1$. If $t = i$, $(\vec{z_j} - \vec{z_i} + \vec{c_{ij}})\vec{s}^T = 0$ holds. Otherwise, with $t \neq i$, $\vec{z_j} - \vec{z_i} + \vec{c_{ij}} = \vec{z_t} - \vec{z_i} - \vec{c_{tj}} + \vec{c_{ij}}$. From Figure 5, $\vec{z_t}$ (or $\vec{z_i}$) is equal to the negation of the summation of the cost vectors along a path from certain node in $V_1$ without any predecessors to $v_t$ (or $v_i$).

First, assume $v_j$ is not a *sink* node. According to Lemma 6, after cancelling the cost vectors for common edges in these two paths, $\vec{z_t} - \vec{z_i} - \vec{c_{tj}} + \vec{c_{ij}}$ will be a summation of the cost vector, or the negation of cost vector, of some distinct edges in $E_1$. Note that each such cost vector is equal to either one of original dependence distance vectors in $E$ or its negation. According to Assumption 3, $abs(\vec{z_j} - \vec{z_i} + \vec{c_{ij}}) < \vec{b}$ holds.

Now, assume $v_j$ is a *sink* node. Note that an M-edge and an L-edge in $E_1$ could *correspond* to the same edge in $E$ in the simplification process of Section 4.1. Suppose $v_t$ stands for $\tilde{L_t}$ and $v_j$ for $\tilde{L_j}^{(k)}$. If the edges $\langle \tilde{L_j}, \tilde{L_i} \rangle$ and $\langle \tilde{L_i}, \tilde{L_j}^{(k)} \rangle$, and the edges $\langle \tilde{L_j}, \tilde{L_t} \rangle$ and $\langle \tilde{L_t}, \tilde{L_j}^{(k)} \rangle$, all correspond to different original edges in $E$, considering the fact that the extension vectors for the original dependence edges in $E$ which are used to compute $\vec{c'_{tj}}$ and $\vec{c'_{ij}}$ may be different, similarly to the argument for the case where $v_j$ is not a *sink* node, we have $abs(\vec{z_j} - \vec{z_i} + \vec{c_{ij}}) < \vec{b} + (0, \ldots, 0, 1)$.

If the edges $\langle \tilde{L_j}, \tilde{L_i} \rangle$ and $\langle \tilde{L_i}, \tilde{L_j}^{(k)} \rangle$ correspond to the same original edge in $E$, say $e_1$, we want to prove that $\vec{d_{e_1}}$ appears at most once in the computation of $\vec{z_j} - \vec{z_i} + \vec{c'_{ij}}$ if we convert back all distances in $E_1$ to those in $E$ by reversing the transformation in Sections 4.1 and 4.2.

- If the edge $\langle \tilde{L_j}, \tilde{L_i} \rangle$ is in the path to compute $\vec{z_t}$, according to the algorithm in Figure 5, the same edge must be in the path to compute $\vec{z_i}$. So they will cancel each other in computing $\vec{z_t} - \vec{z_i}$. The $e_1$ appears only once as the form of $\vec{c'_{ij}}$.

- If the edge $\langle \tilde{L_j}, \tilde{L_i} \rangle$ is in the path to compute $\vec{z_i}$ but not $\vec{z_t}$, $-\vec{z_i} + \vec{c'_{ij}}$ will cancel each other's distance value for that L-edge. The negation of the possible extension vector associated with $e_1$ (in $E_0$) will not be canceled out, however.

- If the edge $\langle \tilde{L_j}, \tilde{L_i} \rangle$ is not in the path to compute either $\vec{z_t}$ or $\vec{z_i}$, the $e_1$ will appear only once as the form of $\vec{c'_{ij}}$.

If the edges $\langle \tilde{L_j}, \tilde{L_t} \rangle$ and $\langle \tilde{L_t}, \tilde{L_j}^{(k)} \rangle$ correspond to the same original dependence edge in $E$, similar argument can be applied such that this original edge appears at most once in the computation of $\vec{z_j} - \vec{z_i} + \vec{c'_{ij}}$.

For all other edges in the computation of $\vec{z_j} - \vec{z_i} + \vec{c'_{ij}}$, similar to the case where $v_j$ is not a sink node, their corresponding original edges in $E$ appear at most once. Therefore, $abs(\vec{z_j} - \vec{z_i} + \vec{c_{ij}}) < \vec{b} + (0, \ldots, 0, 1)$ holds for the case where $v_j$ is a *sink* node.

From Figure 5, it is clear that $\vec{z_j} - \vec{z_i} + \vec{c'_{ij}} \succeq \vec{0}$ holds for $e = \langle v_i, v_j \rangle \in E_1$. According to Lemma 1, $(\vec{z_j} - \vec{z_i} + \vec{c'_{ij}})\vec{s}^T \geq 0$ holds.

## Appendix D: Proof of Lemma 8

Otherwise, we can argment flow value by 1 along the cycle, which has a total negative distance, to get a less objective value for **Problem 4** [13]. Note that augmenting flow value on an edge $\langle v_i, v_j \rangle$ of $G'_2$ which has negative cost is equivalent to reducing the same amount of flow value on the edge $\langle v_j, v_i \rangle$ of $G_2$ which has a positive cost. After augmentation,

- the constraint (24) still holds because for each node in the cycle, flow value is added to both in-edge and out-edge in $G'_2$. If both in-edge and out-edge have the non-negative (or negative) costs, the flow value will be added to both the in-edge and the out-edge of $G_2$. If one of the edges has non-negative cost and the other has negative cost, the total flow value for both in-edges and out-edges will remain the same as before augmentation in $G_2$. In both cases, the constraint (24) holds after augmentation.

- the constraint (25) still holds because for any edge $\langle v_i, v_j \rangle$ in the cycle, if the cost is non-negative, the flow value will be increased for that edge, still non-negative. If the cost for the edge is negative, based on our construction of $G'_2$, the flow value for the edge $\langle v_j, v_i \rangle$ is positive. The flow value augmentation by 1 on the edge $\langle v_i, v_j \rangle$ means to reduce the flow value on $\langle v_j, v_i \rangle$ by 1. The resulted flow value will still be non-negative.

- the objective function value in (23) will be reduced because the cycle has a negative total cost.

This contradicts the fact that $f^*(e)$ is optimal. Thus, there exists no cycles with negative total cost in $G_2'$.

## References

[1] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Improving data locality by array contraction. *IEEE Transactions on Computers*. To appear in vol 53 no. 8, August 2004.

[2] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the Twelfth International Symposium on System Synthesis*, Boca Raton, Florida, November 1999.

[3] Antoine Fraboulet, Karen Kodary, and Anne Mignotte. Loop fusion for memory space optimization. In *Proceedings of the Fourteenth International Symposium on System Synthesis*, pages 95–100, Montreal, Canada, October 2001.

[4] Eddy De Greef, Francky Catthoor, and Hugo De Man. Array placement for storage size reduction in embedded multimedia systems. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Zurich, Switzerland, July 1997.

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

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

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

[8] Junjie Gu, Zhiyuan Li, and Gyungho 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.

[9] Dror Maydan, Saman Amarasinghe, and Monica 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.

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

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

[12] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Naples, Italy, June 2001.

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

[14] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Tomasz Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang-Kang 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.

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

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

[17] Ken Kennedy and Kathryn 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 Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August, 1993.

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

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

[20] Guang R. Gao, Russell Olsen, Vivek Sarkar, and Radhika 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.

[21] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of 2001 ACM Conference on PPOPP*, pages 103–112, Snowbird, Utah, June 2001.

[22] Daniel Cociorva, Gerald Baumgartner, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, Marcel Nooijen, David Bernholdt, and Robert Harrison. Space-time trade-off optimization for a class of electronic structure calculations. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, Berlin, Germany, June 2002.

[23] Geoff Pike and Paul Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of the IEEE/ACM Supercomputing Conference*, Baltimore, MD, November 2002.

```
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

```
$L_1$ : DO $L_{1,1} = l_{11}, l_{11} + b_1 - 1$
        DO $L_{1,2} = l_{12}, l_{12} + b_2 - 1$
          ...
            DO $L_{1,n} = l_{1n}, l_{1n} + b_n - 1$
...
$L_i$ : DO $L_{i,1} = l_{21}, l_{21} + b_1 - 1$
        DO $L_{i,2} = l_{22}, l_{22} + b_2 - 1$
          ...
            DO $L_{i,n} = l_{2n}, l_{2n} + b_n - 1$
...
$L_m$ : DO $L_{m,1} = l_{m1}, l_{m1} + b_1 - 1$
        DO $L_{m,2} = l_{m2}, l_{m2} + b_2 - 1$
          ...
            DO $L_{m,n} = l_{mn}, l_{mn} + b_n - 1$
```

(a)

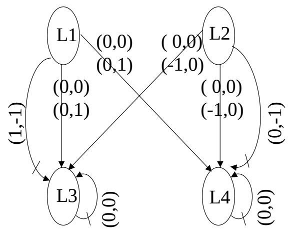
```
$L_1$ : DO $L_{1,1} = l_{11} + p_{11}, l_{11} + p_{11} + b_1 - 1$
        DO $L_{1,2} = l_{12} + p_{12}, l_{12} + p_{12} + b_2 - 1$
          ...
            DO $L_{1,n} = l_{1n} + p_{1n}, l_{1n} + p_{1n} + b_n - 1$
...
$L_i$ : DO $L_{i,1} = l_{i1} + p_{i1}, l_{i1} + p_{i1} + b_1 - 1$
        DO $L_{i,2} = l_{i2} + p_{i2}, l_{i2} + p_{i2} + b_2 - 1$
          ...
            DO $L_{i,n} = l_{in} + p_{in}, l_{in} + p_{in} + b_n - 1$
...
$L_m$ : DO $L_{m,1} = l_{m1} + p_{m1}, l_{m1} + p_{m1} + b_1 - 1$
        DO $L_{m,2} = l_{m2} + p_{m2}, l_{m2} + p_{m2} + b_2 - 1$
          ...
            DO $L_{m,n} = l_{mn} + p_{mn}, l_{mn} + p_{mn} + b_n - 1$
```

(b)

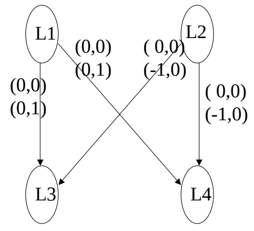
Figure 2: The loop nests before and after loop shifting

```
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)

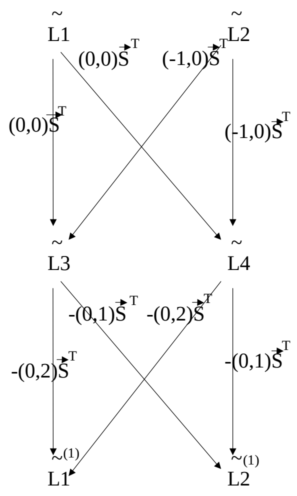


(b)

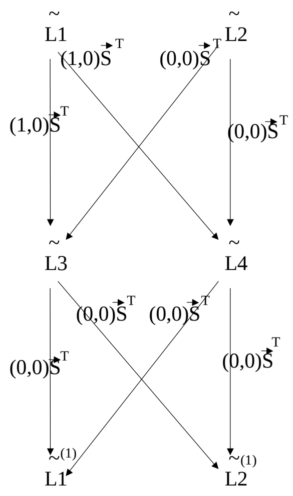


(c)

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



(a)



(b)

Figure 4: The transformed graphs ($G_1$ and $G_2$) for Figure 3(c) ($\vec{s} = (JN-1, 1)$)

```
Input: $G_1 = (V_1, E_1)$.
Output: The lexical height vector $\vec{z_j}$ for each node $v_j \in V_1$.
Procedure:
    for (each node $v_j$ in $V_1$ following a topological order) do
        if ($v_j$ has no predecessors) then
            $\vec{z_j} := \vec{0}$
        else
            $\vec{z_j} := lex\_max\{\vec{z_i} - \vec{c'_{ij}} | \forall \vec{v_i}$ which is a predecessor of $\vec{v_j}\}$
        end if
    end for
```

Figure 5: Computing the lexical height vector for each node in $G_1$

```
Input: $G_2 = (V_2, E_2)$.
Output: Optimal solution for Problem 4 with flow values $x$.
Procedure:
  Set $x_{ij} := 0 (\forall \langle v_i, v_j \rangle \in E_2)$, $\pi_i := 0$ and $e_i := h(v_i)$ $(\forall v_i \in V_2)$.
  Set $\Delta := max\{|e_i| | \forall v_i \in V_2\}$.
  // Initially, all edges in $E_2$ are nonabundant edges.
  while (the residual network $G'_2$ contains a node $v_i$ with $e_i > 0$) do
    if ($max\{e_i | \forall v_i \in V_2\} \leq (\Delta/(8|V_2|))$) then
      $\Delta := max\{e_i | \forall v_i \in V_2\}$.
    end if
    for (each nonabundant edge $\langle v_i, v_j \rangle$) do
      if ($x_{ij} \geq 8\Delta|V_2|$) then
        Designate edge $\langle v_i, v_j \rangle$ as an abundant edge.
      end if
    end do
    // An abundant component is a strongly connected component of $G_2$ with all edges being abundant edges.
    // Imbalance property states that for each abundant component, each nonroot node has a zero imbalance
    // and a root node can have an excess or a deficit.
    Update abundant components and reinstate the imbalance property.
    while ($G'_2$ contains a node $v_k$ with $|e_k| \geq (|V_2| - 1)\Delta/|V_2|$) do
      Select a pair of nodes $v_k$ and $v_l$ satisfying the property that (i) either $e_k > (|V_2| - 1)\Delta/|V_2|$ and $e_l < -\Delta/|V_2|$,
        or (ii) $e_k > \Delta/|V_2|$ and $e_l < -(|V_2| - 1)\Delta/|V_2|$.
      Considering reduced costs $\vec{c_{ij}}\vec{s}^T - \vec{\pi_i}\vec{s}^T + \vec{\pi_j}\vec{s}^T$ as edge lengths, compute shortest path distance $d(.)$
        in $G'_2$ from node $v_k$ to all other nodes.
      $\pi_i := \pi_i - d_i, \forall v_i \in E_2$.
      Augment $\Delta$ units of flow along the shortest path in $G'_2$ from node $v_k$ to $v_l$.
    end do
    $\Delta := \Delta/2$.
  end do
```

Figure 6: Enhanced capacity scaling algorithm

```
Input: A collection of loop nests and its LDG $G = (V, E)$.
Output: A set of optimal shifting vectors $\vec{p_i} (i = 1, \ldots, m)$, and conditions to be checked at run time.
Procedure:
  Verify Assumptions 1-3 and generate necessary condition for Assumption 3.
  Transform the original Problem 0 successively to Problem 3.
  Formulate Problem 4 and solve it using existing network-flow algorithms [13].
  Compute an optimal solution for Problem 3 based on Formula (28).
  Compute an optimal solution for Problem 2 based on Formulas (18) and (20).
  Compute an optimal solution for Problem 1 (and Problem 0) based on Formulas (14) and (15).
  Compute the final condition, under which the transformed code will be executed, by combining all conditions for
    Assumption 3 and Problems 3 and 4.
```

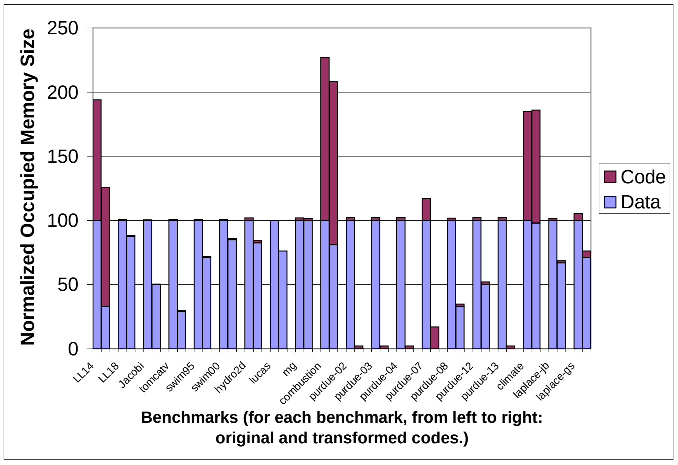
Figure 7: Algorithm 1: computing optimal shifting vectors

```
REAL*8 $ZA0(2:KN), ZA1(1:2), ZB0(2:JN), ZB1(1:JN)$
...
DO $K = 2, KN+1$
  DO $J = 2, JN$
    $T_1 = MOD((JN-1)*(K-2)+J-1, JN)+1$ /* for old $ZB'(J,K)$ */
    $T_2 = MOD((JN-1)*(K-3)+J-1, JN)+1$ /* for old $ZB'(J,K-1)$ */
    $T_3 = MOD((JN-1)*(K-2)+J-2, 2)+1$ /* for old $ZA'(J-1,K)$ */
    $T_4 = MOD((JN-1)*(K-2)+J-1, 2)+1$ /* for old $ZA'(J,K)$ */
    IF $(K.EQ.2)$ THEN
        $ZB1(T_1) = ZQ(J-1,K) + ZZ(J,K)$
    ELSE IF $(K.EQ.(KN+1))$ THEN
        $ZA1(T_4) = ZP(J-1,K) + ZR(J-1,K-2)$
        IF $(J.EQ.2)$ THEN
            $ZP(J,K-1) = ZP(J,K-1) + ZA1(T_4) - ZA0(KN) - ZB1(T_2) + ZB0(J)$
            $ZQ(J,K-1) = ZQ(J,K-1) + ZA1(T_4) + ZA0(KN) + ZB1(T_2) + ZB0(J)$
        ELSE
            $ZP(J,K-1) = ZP(J,K-1) + ZA1(T_4) - ZA1(T_3) - ZB1(T_2) + ZB0(J)$
            $ZQ(J,K-1) = ZQ(J,K-1) + ZA1(T_4) + ZA1(T_3) + ZB1(T_2) + ZB0(J)$
        END IF
    ELSE
        $ZA1(T_4) = ZP(J-1,K) + ZR(J-1,K-2)$
        $ZB1(T_1) = ZQ(J-1,K) + ZZ(J,K)$
        IF $(J.EQ.2)$ THEN
            $ZP(J,K-1) = ZP(J,K-1) + ZA1(T_4) - ZA0(K-1) - ZB1(T_2) + ZB1(T_1)$
            $ZQ(J,K-1) = ZQ(J,K-1) + ZA1(T_4) + ZA0(K-1) + ZB1(T_2) + ZB1(T_1)$
        ELSE
            $ZP(J,K-1) = ZP(J,K-1) + ZA1(T_4) - ZA1(T_3) - ZB1(T_2) + ZB1(T_1)$
            $ZQ(J,K-1) = ZQ(J,K-1) + ZA1(T_4) + ZA1(T_3) + ZB1(T_2) + ZB1(T_1)$
        END IF
    END IF
  END DO
END DO
```

Figure 8: The transformed code for Figure 3(a) after memory reduction

Table I: 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 primality 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

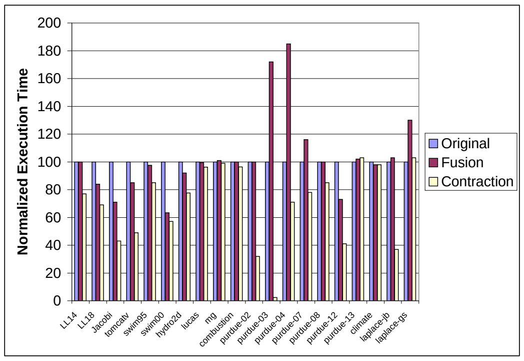


Figure 10: Performance comparison with the original inputs