Locality-Conscious Nested-Loops Parallelization

Saeed Parsa and Mohammad Hamzei

To speed up data-intensive programs, two complementary techniques, namely nested loops parallelization and data locality optimization, should be considered. Effective parallelization techniques distribute the computation and necessary data across different processors, whereas data locality places data on the same processor. Therefore, locality and parallelization may demand different loop transformations. As such, an integrated approach that combines these two can generate much better results than each individual approach. This paper proposes a unified approach that integrates these two techniques to obtain an appropriate loop transformation. Applying this transformation results in coarse grain parallelism through exploiting the largest possible groups of outer permutable loops in addition to data locality through dependence satisfaction at inner loops. These groups can be further tiled to improve data locality through exploiting data reuse in multiple dimensions.

Keywords: Automatic nested loops parallelization, data locality, loop tiling.

I. Introduction

The aim is to speed up data intensive programs for execution on multiprocessor systems with hierarchical memory. To speed up data-intensive programs, there have been two complementary techniques, namely, data locality optimization and nested loops parallelization. Data locality optimization techniques attempt to maximize the number of data accesses satisfied from the higher levels of the memory hierarchy and decrease the costly off-chip accesses. On the other hand, loop parallelization is aimed at speeding up loop executions by distributing independent loop iterations on multiple processors. Both data locality and loop parallelization can be achieved by applying appropriate loop transformations [1]. However, there is an inherent difficulty in treatment with data in these two techniques. Effective parallelization distributes the computation and its necessary data across different processors, whereas locality places data on the same processor. An integrated approach that combines these two can generate much better results in terms of program speedup than using either technique individually [2].

Multidimensional affine scheduling can successfully model complex sequences of loop transformations as one single optimization step [3]. This paper presents a unified approach based on the polyhedral model to achieve multidimensional affine scheduling to increase parallelism and improve data locality for nested loops. The polyhedral model has enabled more complex loop nests to be handled and facilitate composition and application of multiple transformations as a multidimensional schedule [3]. To obtain an appropriate schedule, in our proposed method, the dependency satisfaction is moved to the inner levels in the transformed space to the furthest extent possible. On the other hand, to obtain tileable iteration space, we are interested in obtaining fully permutable groups of loops. As a result, we obtain the largest possible

Manuscript received Mar. 20, 2013; revised Aug. 26, 2013; accepted Sept. 06, 2013.

Saeed Parsa (phone: +98 912 100118, parsa@iust.ac.ir), and Mohammad Hamzei (hamzei@iust.ac.ir) are with the School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

http://dx.doi.org/10.4218/etrij.14.0113.0266

groups of outer fully permutable loops while satisfying dependencies at inner loops as much as possible.

The main contributions of this paper are as follows:

- We propose a unified approach for achieving coarse grain parallelism and data locality improvement simultaneously.
- To overcome the combinatorial nature of the optimization search space, we propose an objective function such that the multidimensional schedule could be determined dimension by dimension.
- We separate the concept of the ratio of tile sides from the concept of the actual size to identify the shape of a tile. The former is determined in such a way as to minimize tile inter-communication, and the latter is determined according to the amount of data that fits in the local cache.

The remaining parts of this paper are organized as follows. In section II, related work is discussed. Section III presents the underlying polyhedral model on which our approach is based. Then, section IV shows the overall steps of a loop transformation framework in the polyhedral model. Section V, explains our proposed algorithm for nested loop parallelization and locality improvement. Experiment results are shown in section VI. Finally, the conclusion and future works are presented in section VII.

II. Related Work

In this section, we discuss different studies on nested loops parallelization and also data locality optimization. There have been various automatic transformation frameworks that try to improve data locality and/or parallelization using code restructuring techniques, although they mainly have some restrictions that do not allow them to effectively optimize nested loops [2], [4], [5].

A majority of the related works on data locality optimization is based on loop transformations. Wolf and Lam [6] proposed a loop transformation method that unifies the various loop transformations, including loop interchange, loop reversal, and loop skewing as a single unimodular matrix. Iteration space tiling is an important loop-based locality enhancing technique. In tiling, multidimensional arrays that are too big to fit in the cache are logically partitioned into smaller chunks that will fit in the cache. In other words, instead of operating on a given array in row (column) order, tiling enables operations on parts of different dimensions of the array at one time. The tiling objective in the context of locality optimization is to keep the active parts of the arrays in the higher levels of the memory hierarchy as long as possible. Therefore, when a data item is reused, it can be accessed from the faster on-chip memory instead of the slower off-chip memory. In [7] some loop transformations are used to make the iteration space of imperfect nested loops tileable. Xue and Huang [8] proposed a locality-driven tiling technique using unimodular loop transformations and data reuse analysis to maximize the data reuse among tiles and minimize the number of tiled dimensions. Also, the problem of selecting tile shape and size to minimize communication and improve locality has been considered extensively [4], [7], [9]. Note that our approach is different from those presented in these studies. These works mainly focused on data locality optimization for sequential and/or parallel nested loops, whereas our approach focuses on automatic locality conscious nested loops parallelization.

There exists a large body of previous works on loop parallelization for multiprocessor systems [5], [10], [11]. In [5], an automatic parallelization of the tiling nested loops was proposed, in which tiles are scheduled and parallel code is generated based on the wavefront method. Lim and others [11] proposed a new partitioning framework that extracts outer parallel loops that are tileable to maximize the parallelism degree and minimize the synchronization cost. In [12], Feautrier proposed a scheduling algorithm to obtain a desirable multidimensional schedule that minimizes latency. However, this study focused on finding maximum fine-grained parallelism suitable for execution on superscalar architectures. Overall, these parallelization algorithms mainly focus on parallelization and do not consider the impact of data locality in the performance of the transformed code. In contrast, we are interested in achieving simultaneous loop parallelization and data locality improvement.

As mentioned above, previous works generally focused on one of the two complementary factors and did not have a realistic cost model as an objective function guiding program optimization. Considering parallelization along with data locality optimization to increase the effectiveness of programs has only been investigated recently. As far as we know, Pluto [2] is the first practical nested loops automatic parallelizer that takes locality improvement into account. It is based, as is our method, on the polyhedral model and attempts to obtain suitable tiling hyperplanes. Although tiling may improve data locality to a certain extent, specific precaution must be taken when obtaining transformations to make a tileable iteration space that supports minimal inter-tile dependencies [4]. To achieve this, using Pluto has been suggested to work out transformations that minimize the bound on the distance between the source and destination of dependent statement instances. Herein, to minimize the bound on all the dependence distances, an integer linear programming (ILP) formulation considering the loop transformation legality constraints is designed. However, this ILP formulation can obtain only positive scheduling coefficients, while there are certain cases in which negative coefficients lead to optimized parallelization

along with data locality. Also, the order of the scheduling coefficients in the objective function of ILP formulation impacts the obtained solution, and it is a time consuming task to examine all the ordering possibilities. Also, in [13], a unified approach that integrates locality and parallelization optimizations for chip multiprocessors was proposed. In that work, the problem was formulated in a constraint network and used a search algorithm to solve it. The main drawback of that work is the lack of a suitable cost model. Also, since all elements of the multidimensional transformation matrix are computed at the same time (instead of dimension by dimension), it seems that dealing with multidimensional transformations leads to a combinatorial explosion. By comparison, in our work, we have a realistic objective function that can be applied dimension by dimension to obtain the multidimensional transformation matrix.

III. Overview of Polyhedral Model

The polyhedral model provides a powerful abstract representation of nested loops based on linear algebra. Using a polyhedral model to model nested loops iteration spaces and statement instance dependencies, it is possible to simply apply any transformations in geometric spaces to transform the nested loops into any desired form [14]. In contrast with the other models of program representation, such as syntax tree and static single assignment form, a polyhedral model considers statement instances rather than the statements themselves. A statement instance is a dynamic instance of a statement in nested loops for a particular iteration of its enclosing loops. This representation is suitable for aggressive optimizations that often need to consider a representation granularity at the statement instance level. In this model, different types of transformations are represented in the form of the interleaving of different instances of statements. Therefore, it is possible to represent a sequence of different program transformations in the form of one transformation in the polyhedral model. The polyhedral model can be used for parts of the program wherein the execution control flow is statically predictable [15].

The polyhedral model includes sets of statement instances for each statement and set of inter-dependencies amongst instances of statements for each of the two dependent statements in the nested loop. Each instance of a statement, s, represents a point within its iteration domain polyhedron, D_s. The iteration domain for each statement indicates the set of dynamic statement instances for all possible values of the enclosing loop indices. For example, s in Fig. 1(a) is executed for each value of the pair of its enclosing loop indices, $x_s=(i, j)$. The iteration domain, D_s, for the statement, s, is defined in (1).

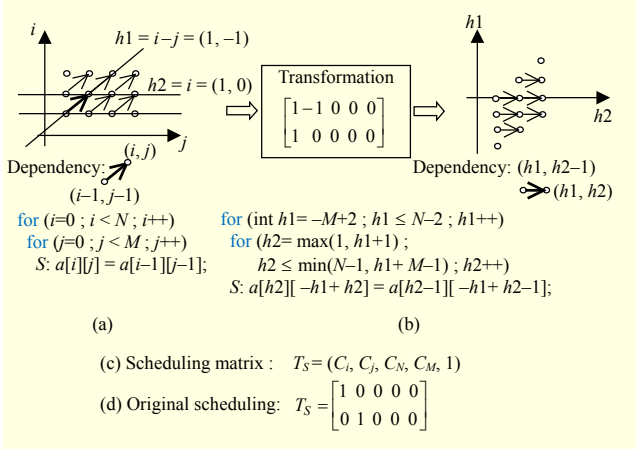


Fig. 1. Sample loop and its transformations.

$$D_s = \{(i, j) \mid 0 \le i \le N-1 \wedge 0 \le j \le M-1\}. \quad (1)$$

In addition to the iteration domain polyhedral, a separate dependence polyhedron, $D_{r,s}$, is built for each pair of dependent statements, r and s. Two statement instances are dependent if they access the same memory location and at least one of them is a write instruction. A data dependence graph (DDG) is used to capture the dependencies among statements. The DDG is a directed acyclic graph (V, E), where V is the set of vertices and E is the set of edges. Each vertex denotes a statement, and each edge indicates dependence from source statement r to target statement s. A set of dependent statement instances of two dependent statements can be formally expressed based on the dependence polyhedron. Therefore, for each edge $e=(r, s)$ of the DDG, there exists a dependence polyhedron, $D_{r,s}$. Dependence polyhedron $D_{r,s}$ is a subset of a Cartesian product of iteration domains for statements r and s. Each integral point inside the dependence polyhedron corresponds to a pair of dependent statement instances. For example, the dependence polyhedron for statement s in Fig. 1(a) is defined as follows:

$$\begin{aligned} D_{s,s} = \{(i, j, i', j') \mid & 1 \le i \le N-1 \wedge 1 \le j \le M-1 \wedge 1 \\ & \le i' \le N-1 \wedge 1 \le j' \le M-1 \wedge i' \\ & = i-1 \wedge j' = j-1\}. \end{aligned} \quad (2)$$

For example, if the integral point (2,2,1,1) resides inside $D_{s,s}$, then the instance of statement s in iteration (2,2) is dependent upon its instance in iteration (1,1).

IV. Transformation Framework

There are three major steps in our nested loop optimization method. In the first step a polyhedral model extractor is invoked to extract the iteration domains polyhedral, the DDG, and its related dependence polyhedron from any sequential C code. We apply Clan and Candle [16] to extract the iteration

http://dx.doi.org/10.4218/etrij.14.0113.0266

domain and the dependence polyhedron from the nested loop, respectively. In the second step, a transformation algorithm is applied to the polyhedral model extracted in the first step, to obtain a transformation suitable for both loop iterations coarse grain parallelism and data locality. In the last step, a polyhedral code generator tool, called Cloog [17], is invoked to generate an optimized code from the iteration domain polyhedral and the transformation matrix, obtained in the first and the second steps, respectively. In the rest of this paper we focus on determining the most suitable transformation for a given nested loop.

As mentioned above, to facilitate nested loop transformations, the polyhedral model of the nested loops is built. This abstract model includes a statement domain polyhedron for each statement within the loop body. The statement domain polyhedron determines the set of statement instances. However, this polyhedron does not specify the order in which these statements are executed. To express the execution order of instances of statement s, a separate scheduling function, φ_s, that determines the execution time of each instance of s as a function of the loop indices is built. Function φ_s for statement s is based on the set of indices of the enclosing loop of s, x_s, and the set of program input parameters, p. Scheduling function φ_s for each statement s is defined in (3).

$$\varphi_s(x_s) = T_s * (x_s, p, 1)^T. \tag{3}$$

In (3), T_S is the $m*(n+p+1)$ matrix of constants and m indicates the number of scheduling dimensions (that is, number of scheduling matrix rows). An example of a scheduling function is presented in Fig. 1(c). In this example, the loop indices vector x_s=(i, j), the parameter vector p=(m, n), and scheduling matrix T_s includes the coefficients for (x_s, p, 1). In Fig. 1(d), a two-dimensional scheduling matrix for the original nested loop in Fig. 1(a) is presented. Multiplying the scheduling matrix shown in Fig. 1(d) with vector $(i, j, m, n, 1)^T$ orders the instances of statement s according to i first and then j, which is the same as the order of execution of statement s instances in the original nested loop, shown in Fig. 1(a).

Each combination of program transformations can be represented as a multidimensional schedule [3]. However, a multidimensional schedule is legal if the relative order of dependent statement instances is not violated. In other words, when transforming a nested loop, the destination of each dependency should be ordered after the source of the dependency. Therefore, a sufficient condition to preserve the original program semantics is to maintain the relative order of such instances. The legality condition for a scheduling function is further formalized below in Theorem 1.

Theorem 1: Legality constraints. A loop transformation is legal if it retains the execution order of dependent statement instances or, to be more precise, transformation T_s for statement s is legal if and only if

$$\forall e \in E \ \forall < x_r, x_s > \in \rho_e \varphi_s(x_s) - \varphi_r(x_r) \geq 0, \tag{4}$$

where ρ_e is the dependence polyhedron related to dependence edge e of the DDG.

If transformation matrix T_s for statement s is a one-dimensional matrix, then the obtained execution time for each instance of s is an integer and all the dependences amongst the instances of s have to be satisfied within a single time dimension. Otherwise, the obtained execution time for each instance of s is a vector that represents the logical time of the execution, where the first dimension of the time is the most significant. In this case, the dependencies might be weakly satisfied until strong satisfaction takes place at a given time dimension. It should be noticed here that a dependency is strongly satisfied if the destination of the dependency is scheduled after its source. To be more precise, dependency e between statements r and s is strongly satisfied at level l of the enclosing loop, if and only if l is the first level at which the condition shown in (5) is met.

$$\begin{aligned} &\forall < x_r, x_s > \in \rho_e \forall k : 1 \leq k \leq l-1 \varphi_s^k(x_s) \\ &-\varphi_r^k(x_r) \geq 0 \ \&\& \ \varphi_s^l(x_s) - \varphi_r^l(x_r) > 0, \end{aligned} \tag{5}$$

where $\varphi_s^k(x_s)$ is the k-th row of the scheduling matrix. Based on this definition, it is said that the dependency at levels before l has been weakly satisfied. If applying schedule matrices for the destination and source of a dependency between statements r and s results in an identical value for first-time dimensions, then the dependency is weakly satisfied at those levels of scheduling matrices. Before the strong satisfaction of the dependencies occurs at level l, all the dependencies must have been weakly satisfied. Once a dependency has been strongly satisfied at level l, no additional constraint is required for legality at levels $l' > l$.

As an example, in Fig. 1, statement s is dependent on itself. After applying transformation matrix T_s to s, the dependence is weakly satisfied at the first level, $h1$, and is strongly satisfied at the second level, $h2$, of the transformed nested loop. For instance, applying T_s, the dependence between instances of s at iterations (1,1) and (2,2) is transformed to iterations (0,1) and (0,2), respectively.

V. Our Approach to Locality Conscious Parallelization

Herein, the aim is to compute a suitable multidimensional scheduling matrix for nested loops to obtain coarse grain parallelism while improving data locality. Cross-iteration

dependencies prevent parallelization, as dependent loop iterations cannot be executed in parallel. On the other hand, each dependency can be considered data reuse because the same data is accessed by the dependent iterations. Data locality can be improved by reducing the reuse distance between accesses to the same data because, when reducing the distance, the accessed data may remain in the cache. Obtaining transformation such that dependencies satisfy at inner loops in the transformed space as much as possible will result in the reduction of reuse distances of the data items accessed within the nested loop body. On the other hand, when transferring dependencies to inner loops, the cross iteration dependencies amongst the nested loop iterations will be moved to inner loops and the outer loops will be prepared for parallel execution. In this way, data locality and parallelism can be achieved simultaneously. To this end, in this section, a new algorithm to obtain the transformation matrix for transferring dependencies to inner loops and parallelizing the outer ones for a given non-perfect/perfect nested loop is presented. Also, to exploit data reuse in multiple dimensions and provide coarse grain parallelism, the algorithm provides a tileable space. Loop tiling is a transformation technique that partitions a loop iteration space into smaller blocks so as to ensure that data used in the loop remains in the cache until it is reused. In addition, tiling for coarse grain parallelism involves partitioning the iteration space into blocks that may be concurrently executed on different processors with a reduced inter-processor communication, as a tile is atomically executed on a processor with communication required only before and after execution. To achieve tileable iteration space, the algorithm attempts to obtain a transformation leading to groups of interchangeable or fully permutable nested loops. A consecutive group of loops within a nested loop is fully permutable if all its components of the dependencies at the related levels are nonnegative. In this case, the execution order of dependent statement instances after any permutation of two loops in the group is respected. Applying scheduling function φ_s, a set of permutable loops at levels $p, p+1, \ldots, p+s-1$ is constructed if and only if

$$\forall e \in E_p \forall < x_r, x_s > \in \rho_e \forall k : p \le k \le p+s-1$$

$$\varphi_{S_s}{}^k(x_s) - \varphi_{S_r}{}^k(x_r) \ge 0, \quad (6)$$

where E_p is the set of dependencies in the DDG that have not been satisfied up to level p–1. Based on this definition, each permutation of the loops at levels $p, p+1, \ldots, p+s-1$ is legal.

It is proven that rectangular tiling could be applied to a group of loops if they are fully permutable [10]. Based on the above discussion, the problem of coarse grain parallelism with locality concerns becomes the problem of obtaining a set of outermost fully permutable groups of nested loops that satisfy dependencies at inner loops as much as possible. Suitable parallelism is obtained through maximizing the parallelization degree for each fully permutable nested loop group in the obtained set.

1. Proposed Algorithm

To obtain a suitable transformation, our proposed algorithm, described in Algorithm 1, attempts to obtain a set of outermost fully permutable nested loops while satisfying dependencies at inner loops. It is shown that a group of fully permutable nested loops of depth n, can be transformed to nested loops that contain at least n–1 degrees of parallelism [10]. In general, for nested loops of depth n, it is not necessarily possible to obtain n–1 parallel loops because all the loops cannot always be transformed to fully permutable loops. To this end, the algorithm attempts to compute a transformation that obtains the largest possible groups of fully permutable nested loops to the extent the legality constraints allow, as described in Theorem 1. After a group of fully permutable loops is constructed, all the dependencies satisfied by the group are excluded from the legality constraints, and the algorithm is repeated until all the dependencies are satisfied. To move the dependencies to the innermost loops, the algorithm attempts to compute transformations at each step, resulting in a fully permutable group that satisfies as few dependencies as possible. To achieve this, the iteration space is partitioned by a number of parallel hyperplanes. The hyperplanes should be of minimal inter-dependencies while each hyperplane should include as many dependent statement instances as possible.

Consider the following non-perfect/perfect nested loop L of depth n, including two dependent statements (r and s) at depth d_r and d_s, respectively. The loop index vector is $x_s = (i_1, i_2, \ldots, i_n)$, where i_k represents the index of L_k (the loop at level k). It is assumed that the nested-loop is normalized and $0 \le i_k \le N_k$. For example, the iteration domain of r can be as shown in (7).

$$D_r = \{(i_1, i_2, \ldots, i_{d_r}) | 0 \le i_1 \le N_1 \wedge \ldots \wedge 0 \le i_1 \le N_{d_r}\}. \quad (7)$$

The dependency polyhedron between r and s, D_{rs}, is represented by the dependent iteration indices (i_r, i_s), where $i_r \in D_r$ and $i_s \in D_s$ and i_r have a relation with i_s based on the dependent iterations. The l throws of the transformation matrix template $T_r : m \times (2d_r + 1)$ for statement r is presented in (8). In (8), C_{lj} are the transformation coefficients for (i_k, N_k, 1) at dimension l of the transformation matrix, where i_k is the loop index vector and N_k is the loop input parameter vector. The transformation matrix template for statement s is similar.

http://dx.doi.org/10.4218/etrij.14.0113.0266

$$T_r = \left[C_{l,i_1} \ldots C_{l,i_{d_r}} C_{l,N_1} \ldots C_{l,N_{d_r}} C_{l,1} \right] \quad (8)$$
$$0 \le l \le m.$$

Algorithm 1. Proposed locality conscious parallelization.
Input: Data dependence graph (*DDG*), dependence polyhedron P_e for all $e \in DDG$
Output: Transformation matrix T
For each dependency edge e in *DDG*
  Build legality constraints LC_e
    if e is nonuniform dependency
      Use Lemma 1 on P_e and eliminate Farkas multipliers
Do
  $LC = \emptyset, i = 0$
  for all e in *DDG*: $LC = LC \cup LC_e$
  $DE = \emptyset$
  do
    Find a hyperplane, H_i, that weakly satisfies constraints in L and strongly satisfies minimum number of dependencies (call these satisfied dependencies D)
    Add H_i as i-th row of T
    $i++$
    $DE = DE \cup D$
    Add independence constraints to LC
while a solution (hyperplane) exists
    $DDG = DDG - DE$
while $DDG != \emptyset$

The main steps of our proposed algorithm for computing transformation matrix T for a given nested loop, L, are as follows.

Step 1. Build the legality constraints.

Step 1.1. For each of the dependent statement instances x_r and x_s of each of the dependent statement r and s, based on Theorem 1, the legality constraint of inequality, represented as greater than or equal to inequality, is obtained. The legality constraint for dependent statements r and s is given in (9).

$$T_s * (x_s, p, 1)^T - T_r * (x_r, p, 1)^T \ge 0. \quad (9)$$

Step 1.2. Legality constraints for nonuniform dependencies are nonlinear. To linearize the constraints, Lemma 1 (described below) is applied.

Step 2. Compute the transformation matrix, dimension by dimension

Step 2.1. Find transformation coefficients (l-th row of T) that weakly satisfy all legality constraints and strongly satisfy the minimum number of dependencies. To this end, search for those values of coefficients in the legality constraints that minimize the number of greater than relations in the greater than or equal to relations of the legality constraints. It should be noted that the greater than inequality in the constraints indicates strong satisfaction of the dependencies. In practice, to minimize the search time overhead, transformation coefficients are assumed to be in a specific range of integers, such as –1 to 1 [3]. A constraint solver can be used to obtain the most suitable solutions. It should be noted that each solution for these constraints can be represented as a hyperplane, as represented in (8). Different instances of the obtained hyperplane (that is, parallel hyperplanes) partition the iteration space into iteration points on different hyperplane instances with maximum intra-dependency and minimum inter-dependency.

Step 2.2. Repeat Step 2.1 as long as new coefficients that are linearly independent from the previous ones can be found (new hyperplane linearly independent from the previous hyperplane).

To this end, the linear independence constraints, obtained based on the method proposed in [2], will be added to the legality constraints.

Step 2.3. If no extra independent hyperplane can be found, exclude all strongly satisfied legality constraints. In this case, if there is no solution for the constraint inequalities, the last column of the transformation matrix should be used to satisfy a dependency from within unsatisfied dependencies by setting the last column of the transformation matrix for the source and destination of the dependency, equal to 0 and 1, respectively. Then, exclude the corresponding constraint of the satisfied dependency and go to Step 2.1 to obtain the next group of hyperplanes, as long as all the constraints are strongly satisfied.

As described above, the inequalities representing the legality constraints obtained for nonuniform dependencies are nonlinear. To linearize a legality constraint, Lemma 1 is applied to its corresponding dependence polyhedron [12].

Lemma 1: Farkas Lemma. Affine form $\delta(x)$ has a nonnegative value at any point inside polyhedron $a_k x + b_k$ if and only if it is a positive affine combination of its faces [6]:

$$\delta(x) \equiv \lambda_0 + \sum_k \lambda_k (a_k x + b_k), \ \lambda_k \ge 0. \quad (10)$$

Applying (10) to the dependence polyhedron, the legality constraint concerned with the dependence polyhedron is transformed into a set of one or more linear equality relations. Using the Fourier-Motzkin elimination method to eliminate the Farkas multipliers, λ_k, in the linear equality relations, linear constraints in terms of transformation coefficients are obtained.

As described in the above algorithm, we look for independent legal hyperplanes that can strongly satisfy the minimum number of dependencies. These hyperplanes constitute a group of fully permutable loops in the transformed space. Following this, to obtain the next group of permutable loops, the satisfied dependencies are excluded from the set of dependencies and this trend continues until all the dependencies are satisfied. The output of the algorithm is a transformation or multidimensional scheduling matrix in which each row contains the coefficients of a transformation

hyperplane.

To increase the parallelism granularity and improve the locality, the transformed iteration space can be tiled. According to the theorem proven in [10], each group can be legally partitioned into rectangular tiles since the transformed space may contain groups of fully permutable loops from applying the proposed algorithm to a given nested loop. Then, a tile can be identified accurately by these hyperplanes and the size of its sides. We compute the size of each side of the tile according to the ratio of its sides and the amount of accessed data in the iteration points residing within the tile. We define the side ratio of each side of the tile as the reverse of the communication cost of that side. The communication cost for each side is computed according to the dependencies carried through the hyperplane that constitute that dimension of the tile. Thus, the side that leads to more communications has a smaller side ratio than one having a smaller number of communications. The actual size of each side is computed based on the size of the local cache of the computational nodes in such a way that the total size of the different pieces of accessed data in each tile can be placed inside a fixed number of local cache lines.

The loops traversing the tiled space have the same properties as those of the nested loop before the tiling. Therefore, these loops are also fully permutable and can be easily parallelized. Finally, the tiles are scheduled for execution on multiprocessors, based on the amount of data items to be commonly accessed through the tiles. The extent of data overlap between two tiles corresponds to their data dependencies plus their RAR (read after read) dependencies. Computing the data overlaps, the tiles are classified into groups that can be executed in parallel, with a minimum synchronization cost. Considering the resultant scheduling, the final parallel loop is generated.

2. Examples

A. Example 1

In this example, the stepwise use of Algorithm 1 to determine a suitable transformation for a simple nested loop is illustrated. Consider the following nested loop, which is also presented in Fig. 1(a):

For (i=0; i<N; i++)
For (j=0; j<M; j++)
S: $a[i][j] = a[i-1][j-1]$.

The iteration domain polyhedron for statement s in the nested loop is

$$D_S = \{(i, j) | 0 \le i \le N-1 \wedge 0 \le j \le M-1\}.$$

The dependence polyhedron for the RAW (read after write) dependency between instances of s, resulting from equality $a[i][j] = a[i-1][j-1]$ for any two iterations, $I = (i, j)$ and $I' = (i', j')$, is

$$\begin{aligned} D_{S,S} = \{(i, j, i', j') \mid & 1 \le i \le N-1 \wedge \le 1 \le j \le M-1 \wedge 1 \\ & \le i' \le N-1 \wedge 1 \le j' \le M-1 \wedge i' \\ & = i-1 \wedge j' = j-1\}. \end{aligned}$$

The legality constraint for this dependency, described in Theorem 1, is obtained as follows:

$$\begin{aligned} & \left(C_i, C_j, C_M, C_N, C_1\right) * (i, j, M, N, 1)^T - \left(C_i, C_j, C_M, C_N, C_1\right) \\ & \qquad * \left(i', j', M, N, 1\right)^T \ge 0 \\ \Rightarrow & \left(C_i, C_j, C_M, C_N, C_1\right) * (i, j, M, N, 1)^T - (C_i, C_j, C_M, C_N, C_1) \\ & \qquad * (i-1, j-1, M, N, 1)^T \ge 0 \\ \Rightarrow & C_i + C_j \ge 0. \end{aligned}$$

Also, to avoid an obvious zero solution for this constraint, a non-zero constraint should be considered: $(C_i \,||\, C_j) \ne 0$.

Aggregating these two constraints, the coefficients $(C_i, C_j, C_M, C_N, C_1)$ of the first transformation hyperplane obtained by our constraint solver is the point (1, –1, 0, 0, 0). This hyperplane strongly satisfies the minimum number of dependencies (that is, obtained constraints). To obtain the next solution, linear independence constraint $C_i + C_j \ne 0$ should be added to the constraints. The constraints are as follows:

$$C_i + C_j \ge 0 \wedge C_i + C_j \ne 0 \wedge (C_i \,||\, C_j) \ne 0.$$

Finding the appropriate value for C_i and C_j, satisfying the above set of inequalities, the coefficients of the resultant hyperplane could be one of the points (1, 0, 0, 0, 0), (1, 1, 0, 0, 0), or (0, 1, 0, 0, 0). All of these hyperplanes strongly satisfy the nested loop statement instances dependencies. Any one of the hyperplanes can be selected as the second hyperplane. Therefore, the resultant transformation matrix can be defined as

$$T_s = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

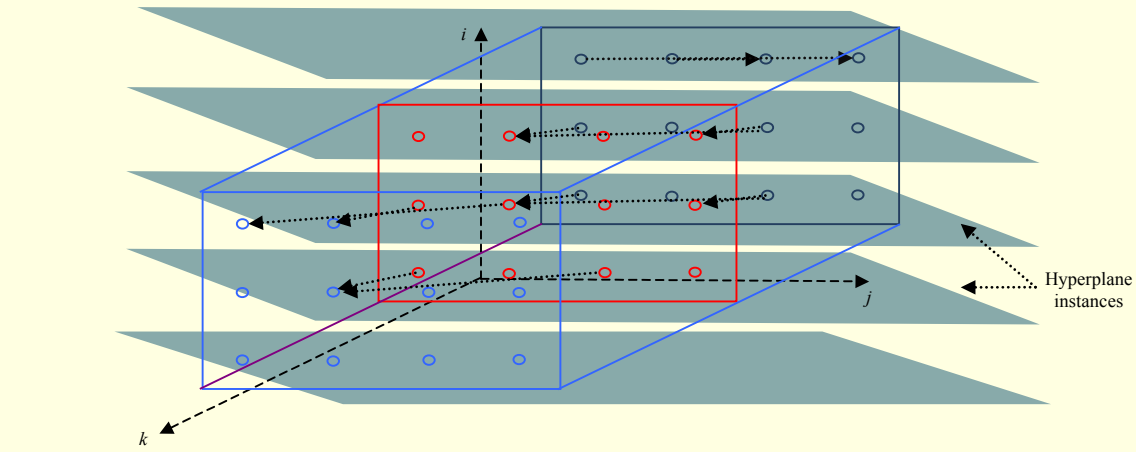
Applying this transformation to the original nested loop results in the nested loop shown in Fig. 1(b). All the iterations of the outer loop of the resultant nested loop can be executed in parallel; additionally, the transformed space is tileable and can be tiled rectangularly.

B. Example 2

Fig. 2(a) shows an example with three dependencies, one of them being a non-uniform dependency. Each dependency and its obtained legality constraint are as follows.

Dependency 1. Uniform RAW dependency:
$a[i][j][k]$ ➔ $a[i-1][j+1][k-1]$.
Obtained constraint: $C_i - C_j + C_k \ge 0$.

http://dx.doi.org/10.4218/etrij.14.0113.0266




```
for (int i=0 ; i ≤ M ; i++)
    for (int j=0 ; j ≤ N ; j++)
        for (int k=0 ; k ≤ O ; k++)
            a[i][j][k]=a[i–1][j+1][k–1]+a[i–1][j–1][k–1]+a[i][N–j][k];
```

(a)

$$T_S = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b)

```
for (int h1= - O ; h1 ≤ M ; h1++)
    for (int h2= max(0, h1) ; h2 ≤ min(M+N, h1+N+O) ; h2++)
        for (int h3=max(max(0, h1), h2–N) ; h3 ≤ min(min(M, h2), h1+O) ; h3++)
            a[h3][h2–h3][ –h1+h3]=a[h3–1][h2–h3+1][ –h1+h3–1]+a[h3–1][h2–h3–1][ –h1+h3–1]+a[h3][N–h2+h3][ –h1+h3];
```

(c)

Fig. 2. Examples, (a) original code, iteration space with some dependency instances, and most suitable transformation hyperplane; (b) computed transformation matrix; and (c) transformed code.

Dependency 2. Uniform RAW dependency:
$a[i][j][k] \rightarrow a[i-1][j-1][k-1]$.

Obtained constraint: $C_i + C_j + C_k \geq 0$.

Dependency 3. Nonuniform RAW dependency:
$a[i][j][k] \rightarrow a[i][N-j][k]$.

Nonlinear legality constraint: $2C_j * j - C_j * N \geq 0$.

Applying Lemma 1:

$$2C_j * j - C_j * N \equiv \lambda_0 + \lambda_1 (N - i) + \lambda_2 (N - j) + \lambda_3 (i) + \lambda_4 (j) + \lambda_5 (-2j).$$

Obtained constraint using Fourier-Motzkin elimination method: $C_j \geq 0$.

Dependency 4. Nonuniform WAR (write after read) dependency:
$a[i][N-j][k] \rightarrow a[i][j][k]$.

Obtained constraint is similar to Dependency 3: $C_j \geq 0$.

Non-Zero constraint: $C_i \parallel C_j \parallel C_k \neq 0$.

Aggregation of all constraints:

$$C_i - C_j + C_k \geq 0 \wedge C_i + C_j + C_k \geq 0 \wedge C_j \geq 0 \wedge (C_i \parallel C_j \parallel C_k) \neq 0.$$

Coefficients $(C_i, C_j, C_k, C_M, C_N, C_O, C_1)$ of the first transformation hyperplane obtained by applying Algorithm 1 are (1,0,–1,0,0,0,0), respectively, which do not strongly satisfy any of the dependencies listed above. Adding linear independence constraint $C_i + C_k \neq 0$, the next solution is $C_i = 1$, $C_j = 1$, $C_k = 0$, which satisfies Dependencies 2, 3, and 4. Finally, adding the second solution linear independence constraint, $C_i - C_j \neq 0$, the third solution is $C_i = 0$, $C_j = 1$, $C_k = 1$, which satisfies Dependency 1. The final transformation matrix is shown in Fig. 2(b).

Applying the obtained transformation matrix to the original nested loop, the resultant code reflects the nested loop shown in Fig. 2(c). Clearly, iterations of the outer loop in the transformed space are independent and can execute in parallel. On the other hand, since the dependency satisfactions are moved to the inner loops, the accessed data items may be reused in the cache, which improves the data locality. Also, the loops in the transformed space are fully permutable and can be tiled rectangularly.

Using the state-of-the-art automatic parallelizer and locality optimizer Pluto [2], the resultant transformation matrix is obtained as follows:

$$T_s = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Clearly, maximum parallelism through parallelizing the outer loop cannot be achieved by applying this transformation.

VI. Experiment Results

To evaluate the proposed method, we use Clan [16] to extract the iteration space polyhedron and Candle [16] to obtain the dependence polyhedron. In addition, we utilize Cloog [17] to generate the final code. To evaluate the effectiveness of the proposed method, we carry out experiments with four data-intensive array-based programs, generally used as benchmarks. The programs are executed on an Intel Xeon E5620 workstation with a dual quad-core 2.8-GHz, 32-KB L1 cache, 12-MB L2 cache, 48 GB of memory, and running Linux. Also, to evaluate the impact of our approach on data locality improvement, we use a simulation environment to model a simple multicore processor.

Figure 3 shows the speedup achieved through our approach over a sequential execution of the original program for all benchmarks on the workstation with the configuration described above. We execute the benchmarks on one, two, four, and eight threads to demonstrate the scalability of our proposed algorithm. We observe that the average speedup gained for one, two, four, and eight threads is about 1.38, 2.36, 4.14, and 7.28, respectively. An important observation that can be made based on this result is that the speedup achieved in some cases, including an optimized sequential version of programs, is more than the number of threads or cores. The speedup above the number of executing threads is due to the reduction in the number of off-chip references of programs, that is, the data locality improvement. This indicates the impact of considering parallelism and data locality improvement as two main factors in performance optimization in terms of speedup.

In Fig. 4, the execution time achieved by applying our algorithm is compared to that achieved by applying Pluto. On average, our algorithm results in about a 16% decrease in program execution time, which shows its superiority over Pluto.

So far, we have demonstrated our experiments on a real machine to show the practicality of our proposed approach. However, to assess the capability of our algorithm regarding improvement of data locality within nested loops, we simulate a simple multicore machine. The simulated machine parameters are shown in Table 1. There are two levels of memory in our simulated machine. The first level, L1, is a private cache for each core, and the second level, L2, is a shared cache amongst all the processor cores. The average cache miss reduction percentages of L1 and L2 on the simulated machine are presented in Fig. 5.

The average cache miss reduction considering all benchmarks is 47.5% and 54% for L1 and L2, respectively. The important point is that applying our approach improves data locality for shared data (inter-core data reuse in L2) amongst all cores as well as for each core (intra-core data reuse in L1) independently. The reason is the decrease in the reuse distance achieved by the algorithm and the reuse of data in multiple dimensions of the arrays accessed within the nested loop body via tiling. This means that, in many cases, a processor core brings a data item to the on-chip shared cache space and uses it and then another core reuses the same data item soon after before removing it from the on-chip cache.

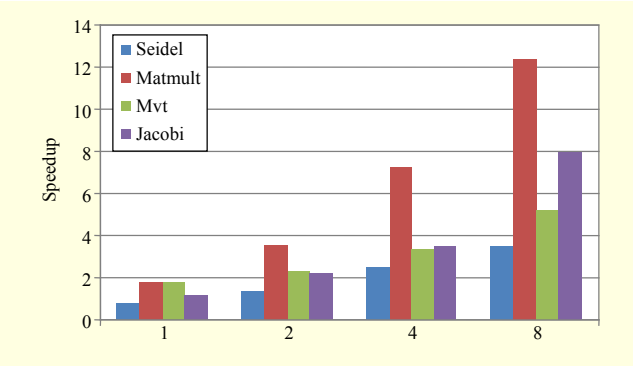


Fig. 3. Speedup on different number of cores.

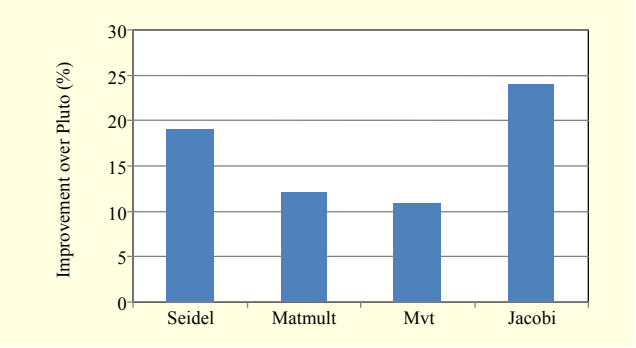


Fig. 4. Improvements in execution time over Pluto.

Table 1. Simulated machine parameters.

Number of cores	L1 size private	L2 size shared	Assciativity L1 & L2	Replacement policy
4	64 KB	1 MB	Full	Strict LRU

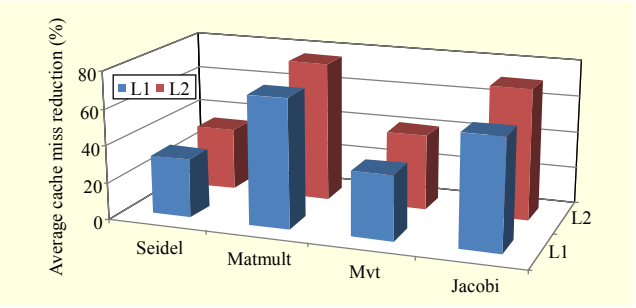


Fig. 5. Cache miss reductions simulation results.

http://dx.doi.org/10.4218/etrij.14.0113.0266

It should be noted that this paper offers a new high-level source-to-source transformation method, which is implemented as a preprocessing phase of compilation. Using compiler optimizations, such as vectorization and pre-fetching, may improve results. Finally, we have seen that the proposed program optimization framework runs fast enough for all these benchmarks and that the compile time is not an issue.

VII. Conclusion

The main problem addressed in this paper was nested loop parallelization along with data locality improvement. The proposed algorithm looks for transformation hyperplanes, resulting in the largest possible groups of outer permutable loops while satisfying dependences at inner levels, as much as possible. Such groups can be tiled to achieve coarse grain parallelism while improving data locality through decreasing the reuse distances and increasing data reuse in multiple dimensions in each tile. Finally, taking the data overlap among tiles into consideration, the tiles could be simply scheduled for parallel execution. Our experiment results demonstrate the effectiveness of our proposed algorithm in optimizing different data-intensive programs. Our future work includes extending the tile scheduling algorithm to take the cache configuration into account.

References

[1] G. Chen and M. Kandemir, "Compiler-Directed Code Restructuring for Improving Performance of MPSoCs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 9, Sept. 2008, pp. 1201-1214.

[2] U. Bondhugula et al., "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, 2008, pp. 101-113.

[3] L.-N. Pouchet et al., "Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time," *ACM SIGPLAN Notices*, vol. 43, no. June 6, 2008, pp. 90-100.

[4] M. Griebl, P. Faber, and C. Lengauer, "Space-Time Mapping and Tiling: A Helpful Combination," *Concurrency Comput., Practice Experience,* vol. 16, no. 2/3, Jan. 2004, pp. 221-246.

[5] S. Lotfi and S. Parsa, "Parallel Loop Generation and Scheduling," *J. Supercomput.*, vol. 50, no. 3, 2009, pp. 289-306.

[6] M.E. Wolf and M.S. Lam, "A Data Locality Optimizing Algorithm," *ACM SIGPLAN Notices*, vol. 26, no. 6, 1991, pp. 30-44.

[7] Y. Song and Z. Li, "New Tiling Techniques to Improve Cache Temporal Locality," *ACM SIGPLAN Notices*, vol. 34, no. 5, 1999, pp. 215-228.

[8] J. Xue and C-H. Huang, "Reuse-Driven Tiling for Improving Data Locality," *Int. J. Parallel Programming*, vol. 26, no. 6, 1998, pp. 671-696.

[9] J. Ramanujam and P. Sadayappan, "Tiling Multidimensional Iteration Spaces for Multicomputers," *J. Parallel Distrib. Comput.*, vol. 16, no. 2, Oct. 1992, pp. 108-120.

[10] M.E. Wolf and M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, Oct. 1991, pp. 452-471.

[11] A.W. Lim, G.I. Cheong, and M.S. Lam, "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication," *Proc. 13th Int. Conf. Supercomput.*, Rhodes, Greece, June 20-25, 1999, pp. 228-237.

[12] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multidimensional Time," *Int. J. Parallel Programming*, vol. 21, no. 6, 1992, pp. 389-420.

[13] O. Ozturk, "Data Locality and Parallelism Optimization Using a Constraint-Based Approach," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, Feb. 2011, pp. 280-287.

[14] A. Cohen, S. Girbal, and O. Temam, "A Polyhedral Approach to Ease the Composition of Program Transformations," *Euro-Par Parallel Process.*, 2004, pp. 292-303.

[15] L.-N. Pouchet, *Iterative Optimization in the Polyhedral Model,* doctoral dissertation, University of Paris-Sud XI, France, 2010.

[16] C. Bastoul, *Extracting Polyhedral Representation from High Level Languages*, Technical report, Paris-Sud University, 2008.

[17] C. Bastoul, "Efficient Code Generation for Automatic Parallelization and Optimization," *Proc. 2nd Int. Symp. Parallel Distrib. Comput.*, Oct. 13-14, 2003, pp. 23-30.

Saeed Parsa received his BS in mathematics and computer science from Sharif University of Technology, Iran, and received his MS and PhD in Computer Science from the University of Salford, England. He is an associate professor of the Computer Science department at Iran University of Science and Technology. His research interests include reverse engineering and parallel and distributed computing.

Mohammad Hamzei received his BS degree from Razi University, Iran, and his MS degree from Iran University of Science and Technology, in 2007 and 2009, respectively, each in Computer Engineering. He is currently working toward his PhD in the Department of Computer Engineering, Iran University of Science and Technology. His research interests are in the areas of parallel and distributed processing, compiler optimization for high performance computing.