Computing and Informatics, Vol. 36, 2017, 566–596, doi: 10.4149/cai_2017_3_566

NESTED-LOOPS TILING FOR PARALLELIZATION AND LOCALITY OPTIMIZATION

Saeed Parsa, Mohammad Hamzei

Department of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
e-mail: {parsa, hamzei}@iust.ac.ir

Abstract. Data locality improvement and nested loops parallelization are two complementary and competing approaches for optimizing loop nests that constitute a large portion of computation times in scientific and engineering programs. While there are effective methods for each one of these, prior studies have paid less attention to address these two simultaneously. This paper proposes a unified approach that integrates these two techniques to obtain an appropriate locality conscious loop transformation to partition the loop iteration space into outer parallel tiled loops. The approach is based on the polyhedral model to achieve a multidimensional affine scheduling as a transformation that result the largest groups of tilable loops with maximum coarse grain parallelism, as far as possible. Furthermore, tiles will be scheduled on processor cores to exploit maximum data reuse through scheduling tiles with high volume of data sharing on the same core consecutively or on different cores with shared cache at around the same time.

Keywords: Nested loops parallelization, loop tiling, data locality, parallel computing

Mathematics Subject Classification 2010: 68N19

1 INTRODUCTION

Loop parallelization without considering the locality of the data to be accessed within the loop body may result in poor performance in terms of speed up. The main barrier against the full utilization of the *chip multi-processors* (CMPs) is the off-chip

memory accesses resulted from cache misses, caused by poor data locality. Therefore, in order to fully utilize CMPs, it is suggested to consider data locality optimization along with loop parallelization [1, 2, 3]. However, data locality optimization could oppose loop parallelization as the data locality tend to place data items on the same processor, while parallelization methods attempt to distribute computations and their required data across parallel processors.

This paper proposes a novel approach to speed up loop execution through loop tiling for coarse grain parallelization along with data locality improvement. Tiling transformation has been widely used to improve data locality in the hierarchical memory, as well as to efficiently execute loop iteration blocks in parallel on multi processors [1, 4, 5]. Applying a tiling transformation, coarse grain parallelism with a reduced inter-tile communication and synchronization could be achieved while the data accesses within each tile or chunk of iterations may be localized to fit into the cache [6]. Loop tiling support in most of the previous works is limited and usually only perfect loop nests are considered [7, 8].

In the approach presented here, we apply a polyhedral pre-transformation as an enabling transformation to help expose tiling opportunities and transform a loop iteration space into a rectangular tilable space. Wolf [9] has proved that nested loops should be transformed into groups of fully permutable loops to be prepared for rectangular tiling. However, any transformation leading to groups of fully permutable loops does not necessarily support tiles with minimized intercommunication and maximized data locality.

To maximize coarse grain parallelism and data locality simultaneously, we compute a transformation matrix to obtain fully permutable loops while moving the loops dependences satisfactions to the inner levels in the transformed space, as much as possible. When moving dependences to the inner loops the reuse distance of the data accessed within the loop body is decreased as each data dependence between statement instances is a data reuse. In this respect, the data locality of the accessed data is enhanced. In addition, moving dependences satisfactions to the inner levels of a nested loop, the outer loops are prepared for *Do-All* parallelization because each cross iteration dependence prevents loop parallelization. As a result, a suitable tilable iteration space will be obtained through finding the largest possible groups of outer fully permutable loops while satisfying dependences at the inner loops. This is our cost function which considers both data locality and parallelism together. To construct the most suitable transformation matrix considering this cost function, we propose a novel geometrical method to compute the most suitable transformation hyperplanes step by step.

In addition to tile shape, the size of the tiles could have an impressive impact on the performance of the tiled code. The ideal is to achieve tiles with minimum inter-tile communication cost and maximum data locality. To achieve this, in the previous works the tile size is determined in such a way that the data footprint of a tile fits the private cache of its underlying processor cores [6]. However, this does not guarantee the minimum intercommunication between tiles. To minimize the inter-tile communication cost, we firstly computes the ratio of the tile sides in such

a way that the side with more communications will be of a smaller size compared with the ones with less communications. Then, to enhance the data locality, the actual sizes of the tiles sides are computed in such a way that the data to be accessed by each tile fits into the private cache of the computational cores.

When iteration space tiling is performed, statement instances in the tiled space are represented by a higher dimensional domains polyhedral involving inter-tile iterators and intra-tile iterators. Inter-tile iterators specify the execution order of tiles that is naturally sequential. In order to schedule tiles for parallel execution, the most common approach is wavefronting transformation which can be applied to the tiled space to create a pipeline-parallel schedule for tile execution [9, 10, 11]. Applying the wavefront transformation the tiled nested loops are modified to expose parallelism at the expense of increasing the reuse distances of data items which results in the poor data locality. To support data locality, we offer a new compile-time tile scheduling algorithm assigning tiles to the processor cores considering the on-chip memory hierarchy of the underlying CMP into account. The scheduling algorithm attempts to exploit maximum data reuse between tiles through scheduling tiles with maximum data sharing on the same core consecutively or on different cores with shared on-chip memory at the same time window. Using the resultant scheduling, the final multi-threaded code is generated.

Briefly, the main contributions of this paper are as follows:

- The design and implementation of efficient nested loops transformation framework for CMPs. In the proposed framework, a unified step by step approach for nested loops parallelization along with data locality optimization is applied.
- A novel algebraic-based method to compute suitable transformation in order to obtain locality optimized tilable iteration space considering outer loop parallelization.
- A new memory hierarchy aware tile scheduling algorithm in order to expose suitable thread-level parallelism.

The remaining parts of this paper are organized as follows: In Section 2, related work is discussed. In Section 3 a brief description of polyhedral models is given. Section 4 describes our loop transformation framework. This section includes our proposed algorithm to obtain tilable nested loop, a new tile scheduling algorithm, an example illustrating the main steps of our loop transformation algorithms and the implementation details of the framework. In Section 5 experimental evaluation demonstrating the efficiency of our loop transformation framework is given. Finally, the conclusion and future works are presented in Section 6.

2 RELATED WORK

Although different studies have been considered nested loops parallelization [9, 12, 4, 13] and data locality enhancement [14, 15, 16] separately, there exists a small

fraction of previous works that consider these two complementary and competing techniques simultaneously.

Previous researches generally focused on one of the two complementary factors and do not consider a scalable and realistic cost model to guide nested loops optimization for CMPs. Increasing the performance of programs through considering coarse grain parallelization along with data locality optimization has been investigated recently. In [3] a constraint network formulation of the problem of parallelization and locality optimization was proposed and a search algorithm was used to solve it. Lack of a suitable cost function is the main drawback of that work. Also, since all elements of the multidimensional scheduling matrix are computed in one step (instead of row by row), it seems that dealing with complex nested loops leads to a search space combinatorial explosion.

Pluto [17] is the state of the art automatic nested loops parallelizer taking data locality optimization into account. Pluto attempts to compute appropriate tiling hyperplanes in the polyhedral model of the given nested loops. Although tiling can improve the data locality, but specific precaution has to be made when computing a transformation matrix to construct a tilable iteration space with minimal inter-tile dependences. To compute an appropriate tiling transformation, Pluto suggests to compute transformations that minimize an upper bound on the reuse distance of dependent statement instances. To minimize the bound on all the reuse distances, an *integer linear programming* (ILP) formulation considering the loop transformation validity constraints is constructed. The main drawback of this ILP formulation is that only positive transformation coefficients could be computed while there are certain cases in which using transformations with negative coefficients, optimized parallelism along with locality could be achieved. For example, in this way any combinations of transformations with negative coefficients including reversal (one of the three main linear loop transformations [9]) are ignored. In addition, every permutation of the transformation coefficients in the objective function of ILP formulation has an impact on the obtained solution and it is a time consuming task to examine all the permutations. In our work we offer a simple realistic cost function that can be used to compute the multidimensional transformation matrix dimension by dimension. Applying the computed transformation to the given nested loops results in appropriate tilable iteration space with minimized inter-tile communications.

On the other hand, most parallelization approaches use wavefronting technique to schedule tiles for parallel execution on multiprocessors [10, 11]. In this way parallel tiled code may suffer from poor data locality and significant load imbalance which results poor scalability on multi processors. Pluto accommodates tile scheduling on multi processors through dynamically allocating tiles to the processors at runtime considering inter-tile dependences [18]. In this paper we propose a new compile-time tile scheduling algorithm taking the tiled code and an on-chip memory hierarchy description as input and schedules tiles for the specified CMP. The algorithm computes the data sharing amongst tiles and classifies them into localized accessed groups that can be executed in parallel with minimum communication and

synchronization cost. In comparison with Pluto scheduling, we offer a compile-time tile scheduling algorithm that considers data reuse amongst tiles in assigning tiles to computational cores to obtain maximum parallelism and data locality together and do not have any run-time overhead. Also in [19] a compile time tile scheduling algorithm based on the Pluto approach is presented which clusters tiles and assigns clusters to computational cores based on the intra and inter-cluster communications. The main drawbacks of that work are that all tiles in a cluster should be assigned to the same processor cores and also do not consider the critical path of the *data dependence graph* (DDG) in the scheduling algorithm.

3 PRELIMINARIES

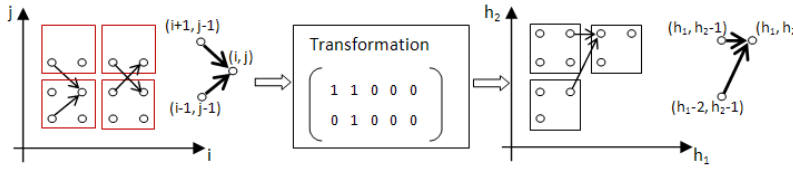
To compute complex transformations, a well-known method is to represent nested loops in the polyhedral model. Polyhedral model provides a powerful abstract algebraic representation of nested loops with the statically predictable control flow, called *static control parts* (SCoP) [20]. In contrast with most program representations that represent an occurrence of a statement in a nested loop only once even if it is executed many times, the polyhedral model considers dynamic statement instances. An instance of a statement is a dynamic occurrence of that statement for a particular iteration of its surrounding loops. Since applying any combination of loop transformations finally results in a new ordering of different instances of statements, then it will be possible to represent a complex sequence of different loop transformations in the form of one transformation matrix in the polyhedral model.

The polyhedral model includes iteration domain for each statement that is a set of the statement instances and dependence polyhedron for each two dependent statements that is a set of interdependences between instances of the two statements in the given nested loop. In fact, iteration domain polyhedron, D_s, for a statement, s, indicates the set of dynamic instances of the statement for all values of the surrounding loops counters. For example, the statement s in Figure 1 a) is executed for each value of its enclosing loop indices, $x_s = (i, j)$. The iteration domain, D_s, for the statement, s, is:

$$D_s = \{(i, j) \mid 0 \leq i \leq M, 0 \leq j \leq N\}.$$

A convenient way to represent the statement dependences is the DDG. DDG is a directed acyclic graph $G(V, E)$, where V is the set of vertices and E is the set of edges. In DDG, each program statement is represented using a vertex, and the dependences between instances of two statements are represented by an edge. For each edge $e = (r, s)$ of DDG there exists a dependence polyhedron, $D_{r,s}$. Dependence polyhedron, $D_{r,s}$, is built for each pair of dependent statements, r and s. Each point in the dependence polyhedron, $D_{r,s}$, represents a dependence between two instances of the statements, r and s.

For example, the dependence polyhedron for the *read-after-write* (RAW) dependence between instances of statement s in Figure 1 a), resulted from equality relation




```
for (i=0; i<=M; i++)
  for (j=0; j<=N; j++)
    S: A[i][j]=A[i-1][j-1]+A[i+1][j-1];
```

(a)

```
for (h1=-N; h1<=M; h1++)
  for (h2=Max(0,h1); h2<=Min(M,h1+N); h2++)
    S: A[h2][-h1+h2]=A[h2-1][ -h1+h2-1]+A[h2+1][ -h1+h2-1];
```

(b)

Figure 1. A sample loop and its transformations

$a[i][j] = a[i-1][j-1]$ for any two iterations $I = (i, j)$ and $I' = (i', j')$ is:

$$D_{s,s} = \{(i, j, i', j') \mid 0 \leq i \leq M, 0 \leq j \leq N, 0 \leq i' \leq M, 0 \leq j' \leq N, \\ i' = i - 1, j' = j - 1\}.$$

For example, the point $(2, 2, 1, 1)$ resides inside the dependence polyhedron $D_{s,s}$. It means that the statement instance s in iteration $(2, 2)$ is dependent on the statement instance in iteration $(1, 1)$ of the nested loop.

As mentioned above, statement domain polyhedron describes the set of dynamic statement instances for each statement. However, the dependence polyhedron does not specify the execution order of the statements instances. Originally, the execution of the statement instances in the nested loops follows lexicographic order of their surrounding loops counters. A convenient way to specify the order in which the statement instances have to be executed is to give a timestamp to each statement instance. Instances of statements are executed in the increasing order of their associated timestamp. A scheduling function $Time_s(x_s)$ determining the timestamp of statement s instances as a function of its surrounding loops counters (iteration vector) is built to express the execution order of instances of the statement. The function $Time_s(x_s)$ is based on the iteration vector of the s surrounding loops, x_s, and program parameters, p. The scheduling function for each statement s is defined as follows:

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

In relation (1), T_s is an $m \times (n + p + 1)$ matrix, m is the number of scheduling matrix rows (i.e. scheduling dimension). An example of a scheduling matrix is presented in Figure 1. In this example the loop iteration vector x_s is (i, j), the parameter vector p is (M, N) and scheduling matrix T_s includes the coefficients for $(x_s, p, 1)$. Multiplying the scheduling matrix with the vector $(i, j, M, N, 1)^T$, orders instances of the statement s according to $i + j$ first and then j which is the new order of execution of the statement s instances in the transformed nested loop in Figure 1 b).

Through this paper, to illustrates different steps of the nested loops transformation we consider a general non-perfect nested loop L of depth n. For example, the loop iteration vector for statement s at depth d_s is $x_s = (i_1, i_2, \ldots, i_{d_s})$ where i_k represents the index of the loop, L_k, at level k of its enclosing loops. It is assumed that the nested-loop is normalized and $0 \leq i_k \leq N_k$. The iteration domain of s is as follows:

$$D_s = \{(i_1, i_2, \ldots, i_{d_s}) \mid 0 \leq i_1 \leq N_1, \ldots, 0 \leq i_{d_s} \leq N_{d_s}\}.$$

The dependence polyhedron, $D_{r,s}$, between two dependent statements r and s is represented by the dependent iteration indices (x_r, x_s) where $x_r \in D_r$ and $x_s \in D_s$.

As discussed in [17] leaving the nested loops input parameters out of scheduling matrix, allows controlling the loop fusion and distribution. Therefore, we also leave the input parameters out of scheduling matrix. The l^{th} row of the statement s scheduling matrix template T_s: $m \times (d_s + 1)$ for statement s is presented in relation (2). In this relation, $C_{l,j}$ are the scheduling coefficients for the loop index i_j or constant 1 at row l of the scheduling matrix. The scheduling matrix template for the statement r is similar.

$$T_s = [C_{l,i_1}, \ldots, C_{l,i_{d_s}}, C_{l,1}], 0 \leq l \leq m. \tag{2}$$

As discussed above, a scheduling function for a statement assigns a logical execution time to each instance of the statement. Applying a combination of loop transformations on a given nested loop may result in different order of executions for the statements instances within the loop body. Thus, any combination of nested loops transformations can be represented by a set of multidimensional scheduling [20] matrices, one for each statement. The scheduling matrices for all statements can be integrated into a new matrix called nested loop scheduling or transformation matrix. The l^{th} row of a nested loop transformation(scheduling) matrix is defined as follows:

$$T_l = \begin{bmatrix} C^l_{s_1,i_1} & \ldots & C^l_{s_1,i_{d_{s_1}}} & C^l_{s_1} & \ldots & \ldots & C^l_{s_n,i_1} & \ldots & C^l_{s_n,i_{d_{s_n}}} & C^l_{s_n} \end{bmatrix}. \tag{3}$$

In this matrix $C^l_{s_m,i_k}$ is the coefficient for the k^{th} iterator of the statement s_m and $C^l_{s_m}$ is a constant coefficient. A transformation(scheduling) function for a nested loop is defined as:

$$Time(x_{s_1}, \ldots, x_{s_n}) = T * (x_{s_1}, 1, \ldots, x_{s_2}, 1, \ldots \ \ldots, x_{s_n}, 1)^T \tag{4}$$

where x_s is an iterator vector of the statement s. In this paper we use the scheduling or transformation terms, alternately.

Applying the multidimensional scheduling function, the source iteration domain polyhedron is transformed into target iteration domain polyhedron containing the same iteration points, with a new lexicographic order. However, applying a multidimensional scheduling matrix result a valid transformed nested loops if the relative order of dependent instances of statements has been respected. In other words,

the destination of each dependence should be scheduled at a time after the source of the dependence. Therefore, a necessary and sufficient condition for preserving semantic of the original program is to maintain the relative execution order of dependent statements instances. The formal definition of the validity constraints for a scheduling function is as follows:

Lemma 1 (Validity constraints)**.** A loop transformation, T_s, for a statement s, is valid if and only if:

$$\forall e \in E, \forall \langle x_r, x_s \rangle \in \rho_e \colon Time_s(x_s) - Time_r(x_r) \geq 0$$

$$\Rightarrow \forall e \in E, \forall \langle x_r, x_s \rangle \in \rho_e \colon T_s * (x_s, 1)^T - T_r * (x_r, 1)^T \geq 0 \tag{5}$$

where ρ_e indicates the dependence polyhedron of the dependence edge e of DDG.

Proof. As proved in [25], preserving the relative order of dependent statement instances is a sufficient condition to maintain the semantics of the original program. Thus, a schedule preserves the semantics of the original program if it does not change the relative order of dependent statements instances. Each integral point of a dependence polyhedron represents a pair of dependent statements instances. Thus, a schedule is valid if and only if it preserves the relative order of instances represented by each integral point of dependence polyhedron for each pair of dependent statements in DDG. Preserving the relative order of a pair of dependent instances $\langle x_r, x_s \rangle$ means that the time specified by scheduling function, $Time(x)$, for the destination of the dependence, x_s, should be greater than equal to the source of the dependence, x_r, as shown in relation (5). □

When the scheduling matrix is one-dimensional, the obtained execution time for each statement instance will be an integer. In this case, all the dependences between the instances of statements should be satisfied at a single time dimension. Otherwise, the execution time computed for each statement instance will be a time vector which represents the time of execution where the first dimension of the time, which is computed by applying the first row of the transformation matrix, is the most significant (like a logical time (hour, minute, second)). Then, the dependences should be satisfied weakly until their strong satisfaction at a time dimension. It will be in place to note that a dependence is satisfied strongly when for each two dependent instances of the statements the destination of the dependence is scheduled at a time after its source.

Definition 1 (Strong satisfaction)**.** A dependence e between two statements r and s is satisfied strongly at a level l of their surrounding loops, if and only if l is the first level at which the following condition is met:

$$\forall \langle x_r, x_s \rangle \in \rho_e, \forall k, 1 \leq k \leq l-1 \colon Time_s^k(x_s) - Time_r^k(x_r) \geq 0$$

$$\wedge\ Time_s^l(x_s) - Time_r^l(x_r) > 0 \tag{6}$$

where $Time_s^k(x_s)$ is computed by applying the k^{th} row of the scheduling matrix to the s surrounding loops iteration vector x_s.

Based on the definition 1, the dependence e at preceding levels of l has been satisfied weakly. A dependence is weakly satisfied at the first time dimensions if applying the corresponding rows of the scheduling matrices results in identical values for the destination and source of the dependence. All the dependences have to be weakly satisfied at the preceding levels of their strong satisfactions. Once a dependence has been satisfied strongly at a level l, no further validity constraints is necessitated for the next levels. Each dependence should be satisfied strongly at a time dimension. Each possible decision about the dimension that the dependence should be satisfied strongly leads to potentially different solutions.

4 TRANSFORMATION FRAMEWORK

The aim is to speed up the compute-intensive program execution on CMPs. To achieve this, we present an efficient approach to optimize nested loops, considering both data locality improvement and parallelization simultaneously. We divide the problem of optimizing nested loops into two sub-problem of obtaining rectangularly tilable iteration space considering locality and parallelism and of tile scheduling considering the memory hierarchy of the underlying CMP. For the first problem, we proposed a novel nested loop transformation algorithm, described in Section 4.1, transforming nested loops to outer parallel tilable nested loops with localized data accesses. For the second problem, we proposed a new tile scheduling algorithm, described in Section 4.2, taking the memory hierarchy into account to assign tiles to computational cores with respect to the amount of data reuse between tiles.

4.1 Tiling Transformation

In this section, a new algorithm is presented to transform a loop iteration space into a valid and appropriate tilable space. Before describing the algorithm, in Section 4.1.1 a polyhedral approach to build the valid transformations space is described. In order to obtain the most suitable transformation within the valid transformations space, a novel cost function is offered in Section 4.1.2. In Section 4.1.3, our proposed algorithm, using the offered cost function to build the most appropriate valid transformation is presented. The transformation could be applied to transform the iteration space into an outer parallel tilable iteration space with improved data locality. At the end, the problem of selecting the optimal tile size is addressed in Section 4.1.4.

4.1.1 Valid Transformations Space

The extracted polyhedral model including the iteration domains polyhedral and dependences polyhedral for each nested loop are used to build the valid transformation space for the nested loop. Here, as shown in Lemma 1, by valid we mean a transformation which does not violate the order of dependent statement instances within the nested loop body. To build the valid transformation space, the validity constraint relation is constructed for each dependence edge in the DDG.

For instance, applying relation (5) to the dependent instances of the statement, s, of dependence polyhedron $D_{s,s}$ resulted from equality $a[i][j] = a[i-1][j-1]$ in Figure 1 a), the validity constraints will be as follows:

$$(C_i, C_j, C_M, C_N, C_1) * (i, j, M, N, 1)^T - (C_i, C_j, C_M, C_N, C_1) * (i', j', M, N, 1)^T \geq 0,$$

$$(C_i, C_j, C_M, C_N, C_1) * (i, j, M, N, 1)^T - (C_i, C_j, C_M, C_N, C_1)(i-1, j-1, M, N, 1)^T \geq 0$$
$$\Rightarrow C_i + C_j \geq 0.$$

Also the validity constraint for dependent instances of the statement, s, of dependence polyhedron $D_{s,s}$ resulted from equality $a[i][j] = a[i+1][j-1]$ will be as follows:

$$-C_i + C_j \geq 0.$$

The validity constraint relation for non-uniform dependences could be non-linear. In order to linearize a validity constraint relation build for a pair of dependent statements, the Farkas lemma [23] is applied to the statements dependence polyhedron. Using affine form of Farkas lemma, such constraints can be translated into an affine equivalent form [23].

Lemma 2 (Farkas lemma)**.** An affine function $\delta(x)$ has a non-negative value at any point inside the polyhedron $a_k x + b_k \geq 0$, iff it is a positive combination of the polyhedron faces:

$$\delta(x) \equiv \lambda_0 + \sum_k \lambda_k (a_k x + b_k), \lambda_k \geq 0. \tag{7}$$

Applying relation (7) to a non-linear validity constraint results in a set of linear equality relations. Then, using the Fourier-Motzkin elimination method Farkas multipliers, λ_k, in the equality relations will be removed and a set of linear validity constraints in terms of transformation coefficients will be computed.

Any feasible solution satisfying all the linear validity constraints provides a vector of valid transformation coefficients, which can be set as a separate row of the transformation matrix. In this way many different transformation matrices, each representing a valid transformation could be achieved. The difficulty is to obtain the most suitable transformation among all the valid transformations based on the desired performance factors. To resolve the difficulty, we proposed a realistic cost function in [24] that considers both data locality and coarse grain parallelism as the main interacting factors in the performance of the nested loops. In the next

section, our proposed cost function is described. In this paper, we provide a novel geometrical method to obtain the most suitable solution based on the proposed cost function.

It should be reminded that the resultant transformation function is in fact a scheduling function which computes a timestamp for each statement instances within the loop body iterations. To achieve this, a transformation matrix considering the linear validity constraints, is computed. The validity constraints provide inequalities, representing the constraint that the source of dependence should be executed before the destination of the dependence. For instance if the source and destination of a dependence between statement instances of a statement are (i_1, j_1) and (i_2, j_2) then the corresponding validity constraint is $(c_i, c_j)(i_2, j_2) - (c_i, c_j)(i_1, j_1) \geq 0$. Our algorithm begins with computing these coefficients from first row of the matrix, i.e. $(c_i c_j)$, considering the cost function discussed in Section 4.1.2. Applying each row of the transformation matrix to the statement instances within the original nested loop, results in a new execution order for statement instances in the transformed space.

4.1.2 Cost Function

Loop tiling is a loop transformation technique that can be applied to obtain coarse grain parallelism and data locality improvement. Tiling for locality optimization involves partitioning iteration space into smaller chunks to ensure the data that is accessed in a chunk remains in the cache until it is reused. As well, tiling for coarse-grain parallelism involves grouping neighboring iteration points together to build a larger block of computation that may be concurrently executed on parallel processors with a reduced communication and synchronization amongst processors. To prepare the nested loop for tiling, often a complex enabling transformation is required to be applied to the nested loop iteration space. Constructing tilable iteration space, our proposed nested loops transformation algorithm attempts to compute a transformation matrix, leading to groups of fully permutable loop nests. As proved in [9], applying a rectangular tiling to a group of loop nests is valid, if they are fully permutable.

Lemma 3 (Fully permutable loops)**.** Applying a scheduling function $Time_s(x_s)$ to the iteration vector of surrounding loops of statement s, a set of fully permutable loops at levels $p, p+1, , p+s-1$ is constructed if and only if:

$$\forall e \in E_p, \forall \langle x_r, x_s \rangle \in \rho_e, \forall k, p \leq k \leq p+s-1, Time_s^k(x_s) - Time_r^k(x_r) \geq 0 \quad (8)$$

where E_p is the set of dependences in DDG that have not been satisfied up to level $p-1$.

Proof. It is proved in [1]. □

Based on this Lemma, each permutation of the loops at levels $p, p+1, \ldots, p+s-1$ is legal.

Although iteration space tiling can improve the data locality and/or coarsen the granularity of parallelism to a certain level, specific precaution has to be made when computing a transformation to construct a tilable iteration space, supporting minimum inter-tile communications [17]. Loops cross-iteration dependences prevent parallelization because dependent iterations of a loop cannot be executed in parallel. On the other hand, each dependence is a type of data reuse because a same data item is accessed by dependent loop iterations. Reducing the reuse distance amongst accesses to the same data item, the data locality could be improved because by reducing the distance the accessed data may be kept in the cache. To reduce the reuse distance, the nested loop should be transformed in such a way that the dependences satisfactions move to the inner loops, as much as possible. In addition, by moving dependences satisfaction to inner loops, the outer loops cross-iteration dependences will be transferred to inner loops and the outer loops iterations will be prepared to be executed in parallel. In this way, the data locality and coarse-grain parallelism could be achieved, simultaneously. Therefore, the problem of data locality enhancement along with coarse grain parallelization will be reformulated as the problem of constructing outermost fully permutable groups of nested loops that tend to satisfy dependences at inner loops. This is our cost function that considers both data locality and parallelism.

4.1.3 The LCLT Proposed Algorithm

Our proposed *Locality Conscious Loop Transformation* (LCLT) algorithm is presented in Figure 2. The algorithm computes a transformation for a given nested loop that transfers the original nested loops to the largest possible groups of outer fully permutable nested loops while transferring dependences satisfaction to inner loops as far as the validity constraints are not violated. The transformation is defined as a matrix, which is computed row by row. Each row is computed as a transformation hyperplane which projects the nested loop iterations to a distinct dimension in the transformed space. It should be noted that applying some consecutive rows of the transformation matrix to the domain polyhedral of a nested loop result in a group of fully permutable loops, if and only if all the dependences are satisfied at least weakly at the corresponding levels. On the other hand, in order to move dependences to the innermost loops, at each step the algorithm attempts to compute transformation hyperplanes result in strongly satisfaction of as few dependences as possible. As proved in Lemma 4 such a transformation hyperplane coefficients are elements of a vertex or a point on the line or ray in the validity constraints polyhedron that is the intersection of maximum number of its faces. We provide a novel geometrical method for computing optimal transformation hyperplanes as proved in Lemma 4.

The algorithm seeks for the new hyperplane as far as new hyperplane that are linearly independent from the previous ones could be found. Linear independence constraints for a set of obtained hyperplanes are computed based on the method offered in [17] and added to the validity constraints. At the point when no extra

Algorithm 1. **LCLT** (Locality Conscious Loop Transformation)

Input: Data Dependence Graph (DDG), Dependence Polyhedron P_e for each e ∈ DDG
Output: Transformation Matrix T

Step 1: // **Compute validity constraints**
  For each dependence edge e **in** DDG **Do**
    Construct the Validity Constraints VC_e, Using Lemma 1
    If e is a Non-Uniform Dependence **then**
      Use Farkas Lemma on P_e and eliminate Farkas Multipliers
    Set $VC = VC \cup VC_e$ where VC is the set of all validity constraints
Step 2 : // **Compute the transformation matrix,** T
  Set the row number i of the transformation matrix, T, equal to 0
  Repeat
    Set the set of all satisfied constraints, $All\_SC = \emptyset$;
    Step 2.1: // Compute the next set of rows of T, resulting in a fully permutable group of nested loops
      Repeat
        Apply Lemma 3 to find the next hyperplane, H_i, in the non-satisfied validity constraints space, $VC - All\_SC$, **such that** H_i weakly satisfy all constraints in VC and strongly satisfy minimum number of constraints in $VC - All\_SC$;
        If $H_i \neq 0$ **then** // a hyperplane is found
          Set H_i as the i-th row of the transformation matrix, T
          Set the row number $i = i + 1$
          Set the set of satisfied constraints, SC = Those constraints in the set of validity constraints ,VC, which are strongly satisfied by H_i
          Set $All\_SC = All\_SC \cup SC$
          Add independence constraints of H_i to VC
      Until no more solution is found
    Step 2.2 : // Compute the next scalar dimension for the transformation matrix
      If $All\_SC = \emptyset$, **then** // if No dependences are satisfied, i.e no solution is found
        Select two dependent statements, r and s, with minimum number of statement instances Dependences, i.e. the minimum number of points in dependence polyhedron P_e for e=<r,s> in DDG)
        Set a scalar dimension H_i which is a hyperplane with all hyperplane coefficients Cs_i and Cr_i equal to zero, $Cr_1 = 0$ and $Cs_1 = 1$
        Set H_i as the i-th row of the transformation matrix, T
        Set the row number $i = i + 1$
        Set the set of satisfied constraints, SC = Those constraints in the set of validity constraints, VC, which are strongly satisfied by H_i
        Set $All\_SC = All\_SC \cup SC$
        Add independence constraints of H_i to VC
    Set $VC = LC - All\_SC$
Until $\boldsymbol{VC == \emptyset}$ and $\boldsymbol{i == Max\_Dim}$ // all constraints are satisfied and the number of Transformation matrix rows is equal to the maximum number of dimension in the original nested loops

Figure 2. LCLT algorithm

independent hyperplane can be found, computed set of the hyperplanes construct a new group of fully permutable loops in the transformed iteration space. Following this, the satisfied dependences constraints are excluded from the validity constraints to compute the next group of fully permutable loops. This trend goes on until all the dependences are satisfied strongly. In step 2.1 of the LCLT algorithm, the next rows of the transformation matrix are computed. If no row is computed, to eliminate some of the constraints from the validity constraints, in step 2.2 a dependence is satisfied through inserting a new scalar dimension into the transformation matrix. The scalar dimension has zero values for all coefficients except the constant coefficients. The constant coefficients are set to zero and one for the source and destination of the dependence, respectively. In this way, the dependence is satisfied by assigning a timestamp to the destination of the dependence that is greater than the timestamp assigned to the source. The coefficients of a transformation hyperplane computed in each stage constitute a row of a multi-dimensional transformation matrix. Applying these transformation hyperplanes to the domain polyhedral of the original nested loop, a suitable rectangular tilable iteration space will be computed.

For example, aggregating the legality constraints obtained for the nested loop of Figure 1 a), the coefficients $(C_i, C_j, C_M, C_N, C_1)$ of the first transformation hyperplane obtained by our algorithm is $(1, 1, 0, 0, 0)$. This hyperplane strongly satisfies a minimum number of dependences (i.e. only the first dependence):

$$\begin{aligned} C_i + C_j &\geq 0 \\ -C_i + C_j &\geq 0 \end{aligned}$$

The next solution should be linearly independent from the first solution. Finding the appropriate value for C_i and C_j which satisfies the validity constraints, the coefficients of the resultant linearly independent hyperplane will be $(0, 1, 0, 0, 0)$. This hyperplane strongly satisfy the second dependence. As a result, the resultant transformation matrix, T_s, is defined as shown in Figure 1.

Applying T_s to the original nested loop statement domain, results in the nested loop shown in the Figure 1 b). The transformed space is tilable and can be tiled, rectangularly. It should be noticed that in this case the nested loop cannot be transformed to the outer parallel loops as the statement dependences span two dimensions of the 2-dimensional nested loop iteration space [9].

Finding the most suitable solution. Before, our method for obtaining the most suitable transformation hyperplane could be described, some backgrounds about geometrical concepts concerning polyhedral optimization seem to be beneficial. A hyperplane is defined as a $n-1$ dimensional sub-space of n-dimensional space. Precisely, an affine hyperplane is the set of vectors v such that $h.v = c$, for $c \in Z$. Different values for c represent different parallel hyperplanes with the vector h as their normal vector. A hyperplane $h.v = c$ partitions the n-dimensional space into two half-spaces, namely the positive halfspace, $h.v > c$, and negative halfspace,

$h.v < c$. A half-space is represented by an affine inequality. A hyperplane and its two halfspaces are shown in Figure 3 a).

A polyhedron is the intersection of finite number of halfspaces which are the faces of the polyhedron. Therefore, a polyhedron can be represented as a system of affine inequalities, each inequality representing a face of the polyhedron. For example, a polyhedron with m faces in the n-dimensional space is shown in relation (9).

$$P = \{x \in R^n \mid Ax + b \geq 0\} \tag{9}$$

where A is a $m * n$ matrix and b is a vector of m elements.

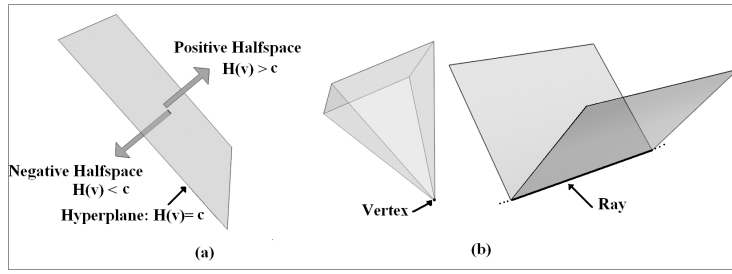


Figure 3. Hyperplane, halfspaces and polyhedron

Based on the concepts mentioned above, the system of affine validity constraints computed in step 1 of the LCLT algorithm constructs a polyhedron called the validity constraints polyhedron. LCLT searches for those values of the coefficients, $H_i = (C_{s1,i_1}, \ldots, C_{s1,i_{d_{s1}}}, C_{s1}, \ldots, C_{sn,i_1}, \ldots, C_{sn,i_{d_{sn}}}, C_{sn})$ in the validity constraints polyhedron that minimize the number of strongly satisfied dependences.

Lemma 4 (LCLT solution point)**.** A point $H_i = (C_{s1,i_1}, \ldots, C_{s1,i_{d_{s1}}}, C_{s1}, \ldots, C_{sn,i_1}, \ldots, C_{sn,i_{d_{sn}}}, C_{sn})$ in the validity constraint polyhedron, minimizing the number of strongly satisfied dependences is a point at the intersection of maximum number of polyhedron faces.

Proof. A polyhedron is a set of affine inequalities and can be represented as a system of affine inequalities:

$$P = \{x \in R^n \mid Ax + b \geq 0\} \equiv \begin{array}{l} a_1 x + b_1 \geq 0, \\ a_2 x + b_2 \geq 0, \\ \ldots \\ a_n x + b_n \geq 0. \end{array}$$

Each point x satisfying an inequality $a_i x + b_i \geq 0$ in the validity constraints polyhedron, $Ax + b \geq 0$, should be a point on the corresponding face, $a_i x + b_i = 0$, or a point in the positive halfspace of the face, $a_i x + b_i > 0$. Each inequality in

the validity constraints polyhedron is a validity constraint indicating the difference between the timestamp of statement instances of the destination and source of the dependence (see Lemma 1). Therefore, the greater than inequality (positive half-space of the face, i.e. $a_i x + b_i > 0$) in the constraints indicates strongly satisfaction of the dependences, because in this case the destination of the dependences scheduled at a time greater than its source in the transformed space. In addition, equal to zero relation for a constraint ($a_i x + b_i = 0$) indicates weakly satisfaction of the related dependence, because in this case the destination and source of the dependence are scheduled to the same time. Hence, a point in the validity constraints polyhedron that does not strongly satisfy a validity constraint $a_i x + b_i \geq 0$, should be on the corresponding face of the polyhedron constructed by the constraint, i.e. $a_i x + b_i = 0$. Therefore, to locate a point minimizing the number of strongly satisfied dependences we should search for a point x that for maximum number of constraints inequalities $a_i x + b_i \geq 0$ results in $a_i x + b_i = 0$. Obviously, such a point is on the maximum number of faces of the polyhedron which is at the intersection of maximum number of intersecting faces of the polyhedron. □

To find a solution point on the intersection of maximum number of faces of a polyhedron, the duality form [26] of the polyhedron is used. It should be noted that each polyhedron can be represented by an alternative dual form in terms of its lines, rays, and vertices. Chernikova algorithm is applied to transfer a polyhedron representation from constraints based representation to dual representation [26]. Vertices, lines and rays are the intersection of at least two faces of the polyhedron. For example, two polyhedral and the solution set of them are given in Figure 3 b).

Based on the above discussion, the solution should be a vertex or a point on a line or a ray at the intersection of the maximum number of faces of the validity constraints polyhedron. If the polyhedron does not have at least one vertex, the ray or line that is at the maximum number of faces is selected as a solution set and any point on the ray or line can be selected as a solution point. Such a point weakly satisfies all constraints as it is in the constraints polyhedron and strongly satisfies minimum number of constraints, as it is at the intersection of maximum number of faces of the polyhedron. If there exist more than one solution, the hyperplane that strongly satisfy minimum number of dependences including Read-After-Read (RAR) dependences is selected. This hyperplane minimize the RAR dependences distances and improve the data locality furthers.

Time complexity. In the following the time complexity of the LCLT algorithm is discussed. The inputs to the algorithm are e and m where e indicates the number of data dependencies and m is the maximum domain dimensionality of the given nested loop. The for loop in step 1 of the algorithm runs for e times which could be as big as $O(e * m^3)$. In the worst case, for each iteration of the loop the Fourier-Motzkin elimination method is applied to eliminate the Farkas multipliers which takes $O(m^3)$. It takes $O(m^2)$ to eliminate a Farkas multipliers from constraints set with m inequalities. Therefore, it takes $O(m^3)$ to eliminate m Farkas multipliers.

Step 2 of the algorithm needs $O(m)$ number of applying Chernikova algorithm [30] to find the optimal solution with respect to Lemma 4. An enhanced implementation of Chernikova algorithm is used for finding a set of vertices and rays of a given polyhedron defined by a system of linear equations and inequalities. Chernikova algorithm takes exponential time $O(2^m)$. However, the Chernikova algorithm is very efficient in practice, although its worst case time complexity is exponential. For example, as shown in [30] for a polyhedron with dimension 8, the execution time of the implemented algorithm is about $0.5s$.

Overall, the worst case complexity appears to be $O(m * 2^m + e * m^3)$. Since our approach relies on Chernikova algorithm, it has a worst-case exponential time complexity $O(2^m)$. However, due to the simple structure of the program polyhedral, our source-to-source code optimizer runs very fast in practice.

4.1.4 Iteration Space Tiling

Applying LCLT algorithm to a given nested loop, the transformed space may contain groups of fully permutable loops. As proved in [9] the iteration space of a group of fully permutable loops can be partitioned into rectangular tiles, legally. A rectangular tiled space could be accurately identified by the computed hyperplanes as tiles sides and the size of each side. In addition to the shape, the tiles size can have significant impact on the performance of the tiled code. To achieve good performance through iteration space tiling, the data items accessed in each tile should reside in the private cache of the processor cores. Computing the optimal size of the tiles is known to be a difficult problem and generally empirical search is used to calculate the optimal size [27]. In order to specify tiles accurately, the tile size will be computed based on our proposed decoupling method to further minimize the communication cost amongst tiles and fit the set of data items accessed within tiles into local caches of the processor cores.

We separate the problem of computing the size of each side of the tile into the problem of computing the size ratio of sides based on the amount of communication across each side and then determining the exact size of each side based on the amount of data accessed within the tile. Separating the size ratio of sides and actual size of the sides can help to consider the two important parameters of inter-tile communication and the amount of accessed data within each tile in the tiling of iteration space. We consider the size ratio of a side as the reverse of the amount of communication across that side to other sides. The amount of communication for each side is obtained based on the number of dependences carried through the hyperplane that constitute that side of the tile. Therefore, the size ratio for sides with more communications will be smaller than the sides with smaller communications. The side ratio, R_i, for side i is computed in such a way that shown in relation (10) (if $com_i = 0$ then set $com_i = 1$):

$$d_i = \frac{MaxCom}{com_i} \Rightarrow R_i = \frac{d_i}{\sum_j d_j}. \tag{10}$$

MaxCom is the maximum communications passed through sides and com_i is the communication of side i. The actual size of each side of the tiles is determined based on the capacity of the private cache of the processor cores in such a way that the average amount of the different accessed data in each tile can be placed inside the private cache. To do this, first the average amount of the different accessed data in each iteration is obtained. Then, the number of all iterations that should be executed in each tile is determined based on the local cache size and the average amount of the different data items accessed in each iteration.

$$AvgDataInIteration = \frac{NumberOfAllDifferentDataAccessed}{NumberOfAllIteration}, \tag{11}$$

$$IntraTileIterations = \frac{CacheSize}{AvgDataInIteration}. \tag{12}$$

Finally, the actual size of each dimension is computed based on the dimension ratio and the number of iterations of the tiles, as shown in relation (13).

$$Size(i) = IntraTileIterations * R_i. \tag{13}$$

4.2 Tile Scheduling

Tile scheduling algorithms are generally aimed at assignment of tiles to the available processors in such a way that the highest possible parallelism in the execution of the tiles is achieved. A general tile scheduling strategy which is used in most previous works is wavefront technique [9, 19]. Tiles within each wavefront are independent from one another and may be executed in parallel, although the wavefronts must be executed in proper sequence to satisfy the loop-carried dependences. Wavefronts can be applied to locate all the tiles that have no interdependences and may execute at the same time in parallel. The main problem of the wavefronting is considering parallelism as the main objective and inattention data sharing between tiles and the possibility of improving data locality. In addition, the number of tiles on each wavefront can be different and may result in imbalanced load on processor cores.

In the rest of this section, we present a different tile scheduling strategy to obtain coarse grain parallelism and improve data locality further. To this end, tiles are statically scheduled for parallel execution on a multiprocessor. The aim of the proposed scheduling algorithm is to exploit the data reuse among tiles through scheduling tiles with high volume of data sharing to be executed consecutively on the same core or on the different cores with the same shared cache at around the same time.

To represent the data reuse between tiles, we use a variation of the DDG, named *data reuse graph* (DRG) which is a DDG including RAR dependences amongst tiles. It should be noted that in contrast to most scheduling algorithms that aimed at minimizing the critical path of the graph, we must also consider the data reuse

amongst the tiles as an important factor in the tiles scheduling. We employ the polyhedral dependence model and construct a dependence polyhedron for all types of data reuse in the tiled space to build our DRG. In addition, each edge of the DRG is labeled by the data reuse degree which is the number of accesses to the same memory locations from the connected pair of tiles.

Our proposed *locality conscious tile scheduling algorithm* (LCTS) is given in Figure 4. LCTS takes a rectangular tiled iteration space and an on-chip memory hierarchy description of its underlying CMP as input and results in a multithreaded code customized for the specified CMP. The proposed tile scheduling algorithm is a type of list scheduling. The algorithm keeps track the tiles in the tree structure of cache memories, from the private cache of each core to the last level of on-chip shared cache amongst all the cores, at each cycle based on the *least recently used* (LRU) cache replacement policy. At each cycle, a node of the DRG is schedulable if all its non-RAR predecessors were being scheduled in the earlier cycles. In addition, the priority of nodes that are the source of at least one non-RAR dependence is higher, as scheduling them make all of their successor nodes schedulable.

The algorithm comprises two main steps. In step 1, all the tiles with non-RAR outgoing edges are scheduled. In fact, a list of schedulable DRG nodes with at least one non-RAR outgoing dependence is considered as ready list. The priority of each node in the ready list is set to the longest distance with the DRG end nodes. Each time, the node with maximum priority is selected and the amount of data sharing between it and tiles in private caches of each core are determined. Afterwards, the selected node is scheduled to the core with maximum data sharing between them. If there are not any data sharing between selected node and the nodes in the private cache of each core, then the data sharing between it and tiles in the next level of the cache hierarchy is considered and this trend continues while empty cores and unscheduled nodes in the ready list are exist.

After scheduling all non-RAR output nodes, in step 2 all the tiles with RAR outgoing edges are scheduled. To schedule the schedulable nodes, a two sided graph is build, one side of which comprises unscheduled schedulable nodes and the other side is provided by all cores with empty cycle. Each node in the graph is connected to all cores with the weight equal to the amount of data sharing between it and tiles in the private cache of each core. A greedy approach is used and repetitively selects the edge with maximum weight and schedule the connected tile to the current cycle of the connected core. Then, the core and all edges connected to that are removed from the graph. This goes on while there exist edges with weight greater than zero. If weights of all edges are equal to zero, weights are set equal to the amount of data sharing between the nodes and tiles in the next level of the cache hierarchy of cores. This trend goes on until all schedulable nodes or cores cycles are scheduled. After that, before going to the scheduling of the next cycle, the schedulable node list will be updated based on the scheduled node in the current cores cycles. After scheduling a cycle of all cores, if there exist cross-cores dependences between tiles scheduled in this cycle and previous cycles, the synchronization table entry for the

Algorithm 2. **LCTS** (Locality Conscious Tile Scheduling)

Input: Data Reuse Graph : G(V, E, W, L), where G is a directed acyclic graph consisting of:
V is the set of rectangular tiles,
E is the set of edges connecting dependet tiles,
W is the set of edge weights representing the amount of data shared between the tiles
L is the set of edge labels indicating RAR and non-RAR dependences
Output: Schedule table, Synchronization table

Cycle = 0
While there exist unscheduled tiles ($G \neq \emptyset$)
Cycle ++ // schedule tiles to the next virtual execution cycle of the processor cores
CandidateTiles = Set of tiles $T_i \in G$ without incoming edges
FreeProcessors = Set of all processors
// Schedule all the tiles $T_i \in$ CandidateTiles
Step 1 Schedule tiles in CandidateTiles with non-RAR outgoing edge
SelectedTiles = tiles $T_i \in$ CandidateTiles, with at least one non-RAR outgoing edge dependency
Step 1.1 Order SelectedTiles according to their longest distance with the end nodes
Foreach tile $T_i \in$ SelectedTiles do
T_i.Priority = The longest distance between T_i and the graph end points
Sort SelectedTiles according to their priority, asscendingly
While SelectedTiles $\neq \emptyset$ and FreeProcessors $\neq \emptyset$
Step 1.2 Compute the amount of data shared between the most prior tile and tiles in each processor (current level of cache), to find the processor with highest amount of data shared with the tile
Select the next tile $T_i \in$ SelectedTiles
Foreach processor $P_i \in$ FreeProcessors do
$SharedData_{i,i} = \sum e(T_i, T_k).weight$,for each tile T_k in the current level of the P_i cache
If $\forall P_i \in$ FreeProcessors: $SharedDate_{i,i} = 0$, then do step 1.2 again for the next level of cache hierarchy
Step 1.3 Schedule the selected tiles
Select (i,j) such that $SharedData_{i,j}$ is maximum and assign T_i to the processor P_i (Schedule[Cycle][P_i] = T_i)
If at least one of the incoming edges of T_i has non-RAR dependence, then mark the current cycle of P_i execution as to-be-synchronized (Synchronization[Cycle][P_i] = true)
Remove T_i and all its connected edges from G
Remove T_i from SelectedTiles and CandidateTiles and P_i from FreeProcessors
Step 2 Schedule tiles with RAR outgoing edge
SelectedTiles = tiles $T_i \in$ CandidateTiles
If SelectedTiles $\neq \emptyset$ and FreeProcessors $\neq \emptyset$
Step 2.1 Construct two sided graph, one side of which comprises the SelectedTiles and the other side is provided by the FreeProcessors
Construct two sided graph, TSG between each $T_i \in$ SelectedTiles and $P_i \in$ FreeProcessors
Foreach tile $T_i \in$ SelectedTiles do
Foreach processor $P_i \in$ FreeProcessors do
$SharedData_{i,i} = \sum e(T_i, T_k).weight$,for each tile T_k in the current level of the P_i cache
Sort graph edges $e(T_i,P_i)$ according to their weight, $SharedData_{i,i}$, asscendingly
If $\forall e(T_i,P_i) \in$ TSG: $SharedDate_{i,i} = 0$, then do step 2.1 again for the next level of cache hierarchy
While SelectedTiles $\neq \emptyset$ and FreeProcessors $\neq \emptyset$
Step 2.2 Schedule the selected tiles
Select next tile-processor edge $e(T_i,P_i)$ and assign tile T_i to the processor P_i (Schedule[Cycle][P_i] = T_i)
Remove T_i and all its connected edges and processor P_i and all its connected edges from TSG
If at least one of the incoming edges of T_i has non-RAR dependence, then mark the current cycle of P_i execution as to-be-synchronized (Synchronization[Cycle][P_i] = true)
Remove T_i and all its connected edges from G
Remove T_i from SelectedTiles and CandidateTiles and P_i from FreeProcessors

Figure 4. LCTS algorithm

core that is the destination of the dependence should be set to be sure that its source is completed.

Applying the proposed algorithm, computational cores accesses the data through private caches as much as possible. On the other hand, considering the cache hierarchy of the CMP into account the processor cores accesses the shared data amongst them at around the same time and as a result maximizing locality of data in shared on-chip memories. In the other word, the algorithm attempt to obtain the maximum parallelization degree and data locality and minimize the off-chip memory accesses.

Finally, the parallel code should be generated based on the scheduling matrix. We use a customized version of polyhedral code generator, Cloog [22], to apply the transformation matrix computed based on the LCLT algorithm to the domain polyhedral of the initial nested loop and generate inter-tile iterator loops. Then, applying the LCTS algorithm the scheduling and synchronization table is obtained. Afterward, the executor threads are constructed and the obtained iterator loops are considered as execution code and the scheduling and synchronization table for each core is passed to that. The executor threads start the execution of the iterations of tiles in the scheduling matrix and if there exist synchronization point synchronize themselves with other threads.

Time complexity. In the following the time complexity of the LCTS algorithm is discussed. The inputs to the algorithm are v and p where v indicates the number of tiles and p is the number of processor cores. The time complexity of step 1 of the algorithm could be as big as $O(Max(v^3, v * p))$. Also, step 2 of the algorithm could be as big as $O(v * (v + p)^2)$. We assume that the number of tiles is much more than the number of processor cores. Overall, the worst case complexity will be $O(v^4)$.

4.3 Example

In this section, we provide a stepwise use of our approach to determine the transformation matrix for a nested loop. Consider the original nested loop presented in Figure 5 a). The iteration domain for the statement s is:

$$D_s = \{(i, j, k) \mid 0 \leq i \leq N-1, 0 \leq j \leq M-1, 0 \leq k \leq O-1\}.$$

The nested loop includes four dependences which two of them are non-uniform dependence. The dependence polyhedron for the uniform *read after write* (RAW) dependence resulted from $a[i][j][k] = a[i-1][j+1][k+1]$ for any two iterations $I = (i, j, k)$ and $I' = (i', j', k')$ is:

$$\begin{aligned} D_{s,s} = \{(i, j, k, i', j', k') \mid & 0 \leq i \leq N-1, 0 \leq j \leq M-1, 0 \leq k \leq O-1, \\ & 0 \leq i' \leq N-1, 0 \leq j' \leq M-1, 0 \leq k' \leq O-1, \\ & i' = i-1, j' = j+1, k' = k+1\}. \end{aligned}$$

```
//(a) Origional nested loop
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        for (k=0; k<O; k++)
            S:a[i][j][k] = a[N-i][j][k] - a[i+2][j+1][k+1] -a[i-1][j+1][k+1];
//(b) Transformed nested loop
for (t1=-M+1;t1<=O-1;t1++)
    for (t2=max(0,-t1);t2<=min(N+M-2,-t1+N+O-2);t2++)
        for (t3=max(max(0,t2-M+1),t1+t2-O+1);t3<=min(min(t2,N-1),t1+t2);t3++)
            a[t3][t2-t3][t1+t2-t3] = a[N-t3][t2-t3][t1+t2-t3] - a[t3+2][t2-t3+1][t1+t2-t3+1] -a[t3-1][t2-t3+1][t1+t2-t3+1];
//(c) Tiled nested loop (Tile size = TS)
for (t1=max(-1,ceil(-M-TS+2,TS));t1<=floor(N+O-2,TS);t1++)
 for (t2=max(0,ceil(TS*t1-O+1,TS));t2<=min(floor(N+M-2,TS),floor(TS*t1+M+TS-2,TS));t2++)
  for (t3=max(max(0,ceil(TS*t1-O-TS+2,TS)),ceil(TS*t2-M-TS+2,TS));t3<=min(min(floor(N-1,TS),t2),t1+1);t3++)
   for (t4=max(max(-M+1,TS*t1-TS*t2),-TS*t2+TS*t3-TS+1);t4<=min(min(min(O-1,TS*t1-TS*t2+TS-1),-TS*t2+N+O-2)
                                                                ,-TS*t2+TS*t3+O+TS-2);t4++)
    for (t5=max(TS*t2,TS*t3-t4);t5<=min(min(min(TS*t2+TS-1,N+M-2),TS*t3+M+TS-1),TS*t3-t4+O+TS-2),-t4+N+O-2);t5++)
     for (t6=max(max(TS*t3,t5-M+1),t4+t5-O+1);t6<=min(min(min(t5,N-1),TS*t3+TS-2),t4+t5);t6++)
      a[t6][t5-t6][t4+t5-t6]=a[N-t6][t5-t6][t4+t5-t6]-a[t6+2][t5-t6+1][t4+t5-t6+1]-a[t6-1][t5-t6+1][t4+t5-t6+1];
//(d) Final nested loop pseudo code of thread #j
for (i=0;i<=ScheduleTable[][j].length;i++) {
  (t1,t2,t3) = ScheduleTable[i][j];
  if (Synchronize[i][j] == true)
    waitForAll(PredecessorTiles(t1,t2,t3));  // wait until signal for all predecessor tiles are not received
  for (t4=max(max(-M+1,TS*t1-TS*t2),-TS*t2+TS*t3-TS+1);t4<=min(min(min(O-1,TS*t1-TS*t2+TS-1),-TS*t2+N+O-2)
                                                               ,-TS*t2+TS*t3+O+TS-2);t4++)
    for (t5=max(TS*t2,TS*t3-t4);t5<=min(min(min(TS*t2+TS-1,N+M-2),TS*t3+M+TS-1),TS*t3-t4+O+TS-2),-t4+N+O-2);t5++)
      for (t6=max(max(TS*t3,t5-M+1),t4+t5-O+1);t6<=min(min(min(t5,N-1),TS*t3+TS-2),t4+t5);t6++)
        a[t6][t5-t6][t4+t5-t6]=a[N-t6][t5-t6][t4+t5-t6]-a[t6+2][t5-t6+1][t4+t5-t6+1]-a[t6-1][t5-t6+1][t4+t5-t6+1];
  signal(Tile(t1,t2,t3));
}
//(e) Scheduling and Synchronization Table
```

Scheduling Table

Cycle	Core 1: (t1,t2,t3)	Core 2: (t1,t2,t3)
1	(-2,1,0)	(-2,2,0)
2	(-1,0,0)	(-1,2,0)
3	(-1,1,0)	(0,0,0)
4	...	...

Synchronization Table

Cycle	Core 1	Core 2
1	false	false
2	false	false
3	false	false
4	...	...

Figure 5. Applying transformation algorithms on an example

The validity constraint for this dependence is as follows:

$$(C_i, C_j, C_k, C_1) * (i, j, k, 1)^T - (C_i, C_j, C_k, C_1)(i', j', k', 1)^T \geq 0$$

$$\Rightarrow (C_i, C_j, C_k, C_1) * (i, j, k, 1)^T - (C_i, C_j, C_k, C_1)(i-1, j+1, k+1, 1)^T \geq 0$$

$$\Rightarrow C_i - C_j - C_k \geq 0.$$

Dependence 2: Uniform WAR dependence, $a[i][j][k] \rightarrow a[i+2][j+1][k+1]$.

Obtained constraint: $2C_i + C_j + C_k \geq 0$.

Dependence 3: Non-uniform RAW dependence, $a[i][j][k] \rightarrow a[N-i][j][k]$.

Non-linear validity constraint: $2C_i * i - N * C_i \geq 0$.

Applying the Farkas lemma to the corresponding dependence polyhedron:

$$2C_i * i - N * C_i \equiv \lambda_0 + \lambda_1(i) + \lambda_2(N - i - 1) + \lambda_3(N).$$

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

Dependence 4: Non-uniform WAR dependence, $a[N-i][j][k] \rightarrow a[i][j][k]$.

Obtained constraint is similar to dependence 3: $C_i \geq 0$.

Also, avoiding obvious zero solution for these constraints, a solution should respect the non-zero constraint:

$$(C_i||C_j||C_j) \neq 0.$$

Aggregating three different validity constraints, the constraint polyhedron is constructed:

$$C_i - C_j - C_k \geq 0, 2C_i + C_j + C_k \geq 0, C_i \geq 0.$$

The first solution found by our method could be any point (C_i, C_j, C_k, C_1) on the line $\{C_j + C_k = 0, C_i = 0\}$ in the constraint polyhedron. It should be explained that this line is the intersection of all faces of the constraints polyhedron and as a result any point on the line does not satisfy any constraint strongly. The point $(0, -1, 1, 0)$ on this line is selected as the first transformation hyperplane coefficients. To obtain the next solution, the algorithm search for a linearly independent solution in the constraints polyhedron. Finding the appropriate linearly independent hyperplane(C_i, C_j, C_k, C_1) in the constraints polyhedron, the resultant hyperplane coefficients can be $(1, 1, 0, 0)$. Finally, the next solution can be $(1, 0, 0, 0)$ which satisfies the last dependence, dependence 1, and is linearly independent from previous solutions. Then, the computed transformation matrix can be defined as:

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

Applying the computed transformation matrix to the original nested loop, the resultant code is shown in Figure 5 b). As it can be seen, the outer loop iterations of the transformed loops are independent and can be executed in parallel. In addition, the transformed iteration space can be tiled rectangularly, as shown in Figure 5 c). After transforming the nested loop and tiling the resultant iteration space, the LCTS tile scheduling algorithm is applied to the tiled space. Applying LCTS algorithm the resultant tile scheduling and synchronization tables are shown in Figure 5 e). The final multi-threaded code is shown in Figure 5 d).

Using Pluto, the state of the art automatic parallelizer and locality optimizer [17], the resultant transformation matrix is computed as follows:

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

As it can be seen, applying this transformation the maximum coarse grain parallelism through outer loops parallelization cannot be achieved.

4.4 Implementation

We use PoCC tools collection [28] to implement our nested loops optimization framework. PoCC is a collection of source code optimization tools and a source-to-source compiler, embedding Clan, Candl [21], Cloog [22], Polylib [29] and some other useful polyhedral compilation libraries.

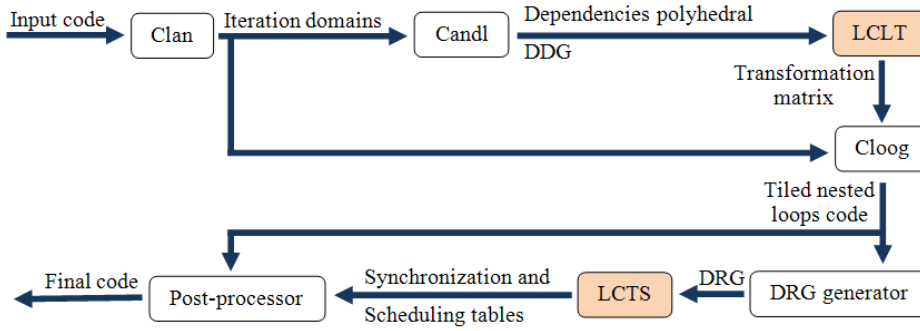


Figure 6. Framework for locality conscious nested loops parallelization

Main components of our proposed optimization framework are shown in Figure 6. In the first step we have applied Clan (Chunky Loop Analyzer) library to extract the iteration domains polyhedral. Clan is a library that translates Static Control Parts (SCoP) of programs written in C or Java into a polyhedral representation. A SCoP is a set of consecutive statements, where loop bounds, array subscripts and conditionals of if statements are affine functions of enclosing loop iterators and input parameters. We invoke Clan to parse the source code of a given nested loop and extract the iteration domain polyhedron for each statement within the SCoP. The output of Clan is a list of matrices, where each matrix represents an iteration domain polyhedron of a statement within the loops body.

In the second step, DDG and its related dependency polyhedral are extracted by Candl (Chunky ANalyzer for Dependencies in Loops). Candl is a library for data dependence analysis of SCoPs. Candl is able to extract the set of dependencies between statement instances from a polyhedral representation of a SCoP obtained in previous step. It takes the SCoP and iteration domains as input and computes the dependence polyhedron for each pair of dependent statements. Using the resultant dependencies polyhedral, DDG of the SCoP can be constructed easily.

Next, our proposed algorithm LCLT, described in Section 4.1.3, is applied to the polyhedral model extracted in the previous steps to compute a suitable transformation matrix. Using the dependences polyhedral and iteration domains, the valid transformation space is constructed based on the Lemma 1. Afterwards, to compute the optimal transformation matrix level by level, the optimal solution at each level is obtained from dual representation of constraints polyhedron which can be computed using Chernikova algorithm implemented in PolyLib library. Polylib

is a library for the manipulation of polyhedral. Polylib uses the Chernikova algorithm to move between the implicit form of a polyhedron (as a set of constraints) and its dual form as a set of lines, rays and vertices. Also, independent constraints is computed based on the method proposed in [1] and its implementation already exists in PoCC. Finally, tile size is deterimind based on the method described in Section 4.1.4.

A customized version of polyhedral code generator tool Cloog (Chunky Loop Generator), converts the initial sequential loop to the desired parallel loop using the iteration domains polyhedral and computed transformation matrix. CLoog is a library for code generation in the polyhedral model. Cloog can scan a union of polyhedral under a lexicographic ordering specified as a transformation matrix. LCLT component computes transformation matrix and provides it to Cloog. On providing the iteration domains along with transformation matrix T, Cloog can generate nested loops which scan the iteration domains in the lexicographic ordering imposed by T for all statements.

In the next step, to construct our DRG we employ the polyhedral dependence model and deduce a polyhedron for data reuse in the tiled domain using Candl. A higher dimensional dependence polyhedron is used to represent data dependences between two dependent statements in the tiled iteration space. Dimensions of intra-tile iterators are projected out from the dependence polyhedron to extract inter-tile dependences in the tiled space. The projection method is performed for all dependence polyhedral using Fourier-Motzkin elimination method implemented in Polylib library. Using the obtained inter-tile dependences, the DRG can be constructed.

In the next step, the proposed tile scheduling algorithm, described in Section 4.2, takes the resultant DRG as input and computes a tile scheduling and synchronization table which determines the execution order of tiles considering the memory hierarchy of the underlying processor architecture. Implementation of this component is based on the pseudo code shown in Figure 4.

Finally, a postprocessor component takes the tiled code generated by Cloog and synchronization and scheduling table obtained in the previous steps as input and generates final parallel code. In fact, the postprocessor removes inter-tile iterator loops from tiled code and adds a new loop with necessary synchronization codes. The newly added loop selects the specified tile at each cycle and executes its intra-tile code. The resultant code may act as the main function of some threads which are attached to different processor cores.

5 PERFORMANCE EVALUATION

In order to evaluate the proposed method, we implement our source-to-source transformation method as a preprocessing phase of compilation. We used Clan [21] to obtain the iteration domains and Candl [21] to extract the dependences polyhedral from the source code. Also, we used Cloog [22] and customized it to generate the final multithreaded code. We carry out experiments with five compute-intensive

array based nested loops generally used as benchmarks to evaluate the proposed method (from polybench benchmark suit [31]). Our evaluation is performed on Intel Xeon E5620 workstation with dual quad-core 2.8 GHz, 32 KB L1 cache, 12 MB L2 cache, 48 GB memory running Linux Ubuntu server 12.04. All versions of the benchmarks are compiled by Intel's ICC 10.1 compiler with '-fast' option. In addition, we used a simulation environment for modeling a simple multiprocessor system with cache hierarchies to evaluate the impact of the proposed method on the data locality enhancement.

Our evaluation starts with a comparison of execution time of four different versions of each code in our benchmark suite on the workstation with 1, 2, 4 and 8 threads. The first one (original) is the original nested loop source code. The second version (LCLT) is the transformed code by applying the LCLT algorithm without tiling. The third version (Tiled-LCLT) is the tiled version of the LCLT optimized version. Finally the fourth one (LCLT-LCTS) is the tiled version of the code optimized by applying LCLT and scheduled by LCTS algorithm. The normalized execution times of different optimized versions of benchmarks with respect to the original version are given in Figure 7. As expected, in average the best case is the LCLT-LCTS versions which are the transformed, tiled and scheduled code by applying all steps of our optimization approach. An important observation that can be made is that as the machine scale out, our LCLT-LCTS version of codes makes better results in comparison with the LCLT and Tiled-LCLT versions which are due to the data locality improvement of tiled code along with coarse grain parallelization and suitable scheduling of tiles. Some important results that can be obtained are as follows:

1. LCLT prepares the iteration space for tiling; without tiling, performance may not be good enough due to ignoring the data reuse in multiple dimension and coarse grain parallelism,
2. due to the overhead of reading from scheduling and synchronization table, the LCLT-LCTS is not the best choice for small number of cores; in this case Tiled-LCLT generates better results and
3. our proposed approach (LCLT-LCTS version) provides a scalable farmework and brings more performance on CMPs with large number of cores; because of exploiting data reuse, parallelism and load balance.

In Figure 8, the improvements in execution time on 8 cores obtained by applying our method (LCLT-LCTS version) are compared with those of Pluto, the state of the art approach in parallelization and locality improvement. In comparison with Pluto, there is about 17 % decrease in the execution time of the programs on average, which shows the superiority of the proposed approach.

On the other hand, to present the impact of the proposed algorithm in locality improvement, the cache-miss reductions gained under our approach on a simple simulated machine for all the benchmarks are shown in Figure 9. Specifically, we simulated a simple multi-core processor that can issue and execute four instructions

#of threads	1					2					4					8				
Benchmark	doitgen	gemver	mvt	syr2k	syrk	doitgen	gemver	mvt	syr2k	syrk	doitgen	gemver	mvt	syr2k	syrk	doitgen	gemver	mvt	syr2k	syrk
LCLT	0.94	0.9	1.04	0.99	0.94	0.83	0.76	0.85	0.91	0.84	0.65	0.41	0.71	0.83	0.72	0.59	0.29	0.67	0.78	0.68
Tiled-LCLT	0.54	0.33	0.53	0.62	0.57	0.52	0.29	0.39	0.49	0.36	0.44	0.19	0.23	0.47	0.29	0.37	0.12	0.19	0.41	0.24
LCLT&LCTS	0.61	0.47	0.6	0.59	0.58	0.5	0.3	0.21	0.42	0.34	0.32	0.14	0.13	0.37	0.24	0.28	0.08	0.09	0.31	0.18

Figure 7. Normalized execution times on different number of cores

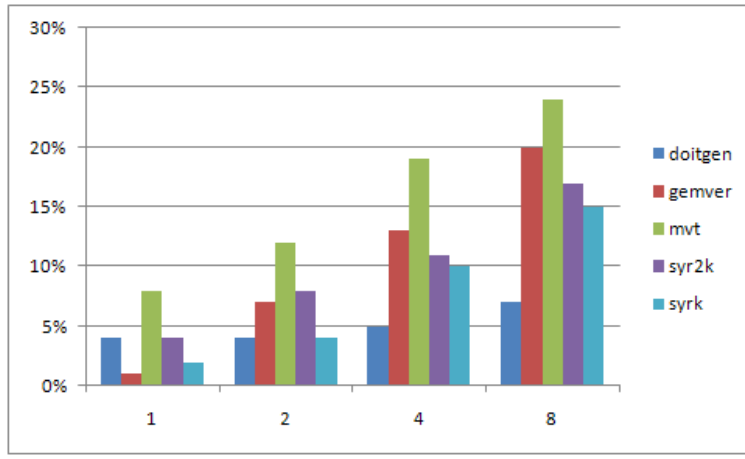


Figure 8. Improvements in execution time over Pluto

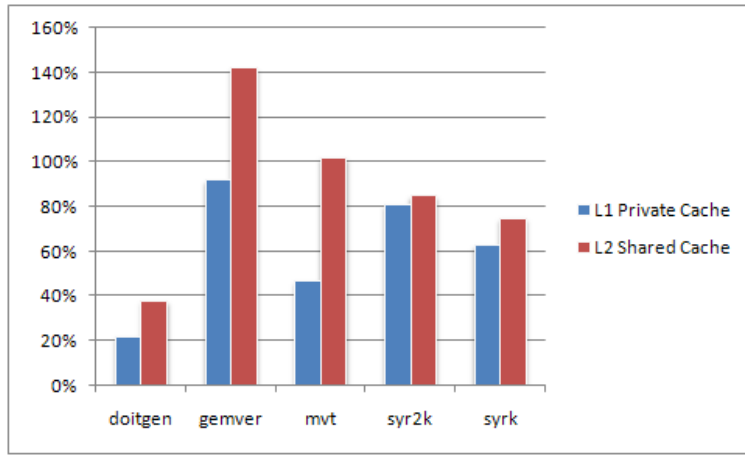


Figure 9. Cache miss reduction in private L1 and shared L2 caches

in parallel. The machine configuration includes private L1 data caches; each is 16 KB and fully associative, and shared L2 cache; 1 MB and fully associative. The average percentages of private L1 and shared L2 cache miss reductions on this machine are shown in the Figure 9. The important point is that by applying the proposed method the data locality enhancement can be achieved through data reuse of shared data items amongst all cores in L2 cache as well as data reuse of private data items in L1 cache of each core. The main reasons summarized as follows:

1. decreasing reuse distances of data items achieved by the LCLT algorithm,
2. exploiting data reuse in multiple dimensions of the arrays via tiling and
3. obtaining locality through tile scheduling by assigning tiles with high volume of data sharing to the same core or different cores with shared cache at around the same time.

6 CONCLUSIONS

This paper presents a new compile-time source-to-source loop transformation approach based on the polyhedral model to address the problem of data locality enhancement along with coarse grain parallelization. In order to maximize the parallelism degree and minimize the reuse distance of data items, the proposed method objective is to obtain a set of outermost fully permutable nested loops that are tilable while moving the dependences satisfaction to the inner loops, as much as possible. Applying the computed transformation following by tiling and memory hierarchy-aware tile scheduling strategy, the coarse-grained parallel nested loops with localized data accesses will be obtained.

REFERENCES

[1] Bondhugula, U. K.: Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model. Ph.D. Dissertation. Ohio State University, Columbus, OH, 2008.

[2] Yemliha, T.—Kandemir, M.—Ozturk, O.—Kultursay, E.—Muralidhara, S. P.: Code Scheduling for Optimizing Parallelism and Data Locality. In: D'Ambra, P., Guarracino, M., Talia, D. (Eds.): Proceedings of the 16th International Euro-Par Conference on Parallel Processing (EuroPar '10): Part I, Springer-Verlag, Berlin, 2010, pp. 204–216.

[3] Ozturk, O.: Data Locality and Parallelism Optimization Using a Constraint-Based Approach. Journal of Parallel and Distributed Computing, Vol. 71, 2011, No. 2, pp. 280–287, doi: 10.1016/j.jpdc.2010.08.005.

[4] Griebl, M.—Faber, P.—Lengauer, C.: SpaceTime Mapping and Tiling: A Helpful Combination. Concurrency and Computation: Practice and Experience, Vol. 16, 2004, No. 1, pp. 221–246.

[5] Xue, J.—Huang, C. H.: Reuse-Driven Tiling for Improving Data Locality. International Journal of Parallel Programming, Vol. 26, 1998, No. 6, pp. 671–696.

[6] Song, Y.—Li, Z.: New Tiling Techniques to Improve Cache Temporal Locality. Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99), New York, 1999, pp. 215–228, doi: 10.1145/301618.301668.

[7] Parsa, S.—Lotfi, S.: A New Genetic Algorithm for Loop Tiling. Journal of Supercomputing, Vol. 37, 2006, No. 3, pp. 249–269, doi: 10.1007/s11227-006-6367-9.

[8] Ramanujam, J.—Sadayappan, P.: Tiling Multidimensional Iteration Spaces for Multicomputers. Journal of Parallel and Distributed Computing, Vol. 16, 1992, No. 2, pp. 108–120, doi: 10.1016/0743-7315(92)90027-k.

[9] Wolf, M. E.—Lam, M. S.: A Loop Transformation Theory and an Algorithm to Maximize Parallelism. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, 1991, No. 4, pp. 452–471, doi: 10.1109/71.97902.

[10] Goumas, G.—Athanasaki, M.—Koziris, N.: An Efficient Code Generation Technique for Tiled Iteration Spaces. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, 2003, No. 10, pp. 1021–1034, doi: 10.1109/tpds.2003.1239870.

[11] Lotfi, S.—Parsa, S.: Parallel Loop Generation and Scheduling. Journal of Supercomputing, Vol. 50, 2009, No. 3, pp. 289–306, doi: 10.1007/s11227-008-0262-5.

[12] Liu, D.—Shao, Z.—Wang, M.—Guo, M.—Xue, J.: Optimal Loop Parallelization for Maximizing Iteration-Level Parallelism. Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09), New York, 2009, pp. 67–76, doi: 10.1145/1629395.1629407.

[13] Pradelle, B.—Ketterlin, A.—Clauss, P.: Polyhedral Parallelization of Binary Code. ACM Transactions on Architecture and Code Optimization, Vol. 8, 2012, No. 4, pp. 1–39, doi: 10.1145/2086696.2086718.

[14] Lam, M. S.—Wolf, M. E.: A Data Locality Optimizing Algorithm. SIGPLAN Notices, Vol. 39, 2004, No. 4, pp. 442–459, doi: 10.1145/113445.113449.

[15] Chen, G.—Kandemir, M.: Compiler-Directed Code Restructuring for Improving Performance of MPSoCs. IEEE Transactions on Parallel and Distributed Systems, Vol. 19, 2008, No. 9, pp. 1201–1214, doi: 10.1109/TPDS.2007.70760.

[16] Ding, W.—Kandemir, M.: Improving Last Level Cache Locality by Integrating Loop and Data Transformations. Proceedings of the International Conference on Computer-Aided Design (ICCAD '12), New York, 2012, pp. 65–72, doi: 10.1145/2429384.2429398.

[17] Bondhugula, U.—Baskaran, M.—Krishnamoorthy, S.—Ramanujam, J.—Rountev, A.—Sadayappan P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. 17th International Conference on Compiler Construction (CC '08), Hungary, April 2008, pp. 132–146, doi: 10.1007/978-3-540-78791-4_9.

[18] Baskaran, M.—Vydyanathan, N.—Bondhugula, U.—Ramanujam, J.—Rountev, A.—Sadayappan P.: Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. Proceedings of the 14th

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09), New York, 2009, pp. 219–228, doi: 10.1145/1594835.1504209.

[19] Liu, J.—Zhang, Y.—Ding, W.—Kandemir, M.: On-Chip Cache Hierarchy-Aware Tile Scheduling for Multicore Machines. Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11), Washington, 2011, pp. 161–170.

[20] Pouchet, L.—Bastoul, C.—Cohen, A.—Cavazos, J.: Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08), New York, 2008, pp. 90–100, doi: 10.1145/1375581.1375594.

[21] Bastoul, C.: Extracting Polyhedral Representation from High Level Languages. Technical Report at Paris-Sud University, Paris, 2008.

[22] Bastoul, C.: Efficient Code Generation for Automatic Parallelization and Optimization. Proceedings of the Second International Conference on Parallel and Distributed Computing (ISPDC '03), Washington DC, 2003, pp. 23–30, doi: 10.1109/ispdc.2003.1267639.

[23] Feautrier, P.: Some Efficient Solutions to the Affine Scheduling Problem: II. Multidimensional Time. International Journal of Parallel Programming, Vol. 21, 1992, No. 6, pp. 389–420.

[24] Parsa, S.—Hamzei, M.: Locality-Conscious Nested-Loops Parallelization. ETRI Journal, Vol. 36, 2014, No. 1, pp. 124–133, doi: 10.4218/etrij.14.0113.0266.

[25] Bernstein, A.: Analysis of Programs for Parallel Processing. IEEE Transactions on Electronic Computers, Vol. 15, 1966, No. 5, pp. 757–763, doi: 10.1109/pgec.1966.264565.

[26] Loechner, V.—Wilde, D. K.: Parameterized Polyhedra and Their Vertices. International Journal of Parallel Programming, Vol. 25, 1997, No. 6, pp. 525–549, doi: 10.1023/A:1025117523902.

[27] Pouchet, L.: Iterative Optimization in the Polyhedral Model. Ph.D. Dissertation. France University of Paris-Sud XI, Paris, 2010.

[28] Pouchet, L.—Bastoul, C.—Bondhugula, U.: PoCC: The Polyhedral Compiler Collection. Availaible on: `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

[29] Loechner, V.: PolyLib: A Library for Manipulating Parameterized Polyhedra. Availaible on: `http://icps.u-strasbg.fr/loechner/polylib`.

[30] Verge, H. L.: A Note on Cherniakova's Algorithm. Technical Report RR-1662. INRIA, 1992.

[31] Pouchet, L.: Polybench: The Polyhedral Benchmark Suite. Availaible on: `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

Saeed Parsa received his B.Sc. in mathematics and computer science from Sharif University of Technology, Iran, his M.Sc. degree in computer science from the University of Salford in England, and his Ph.D. in computer science from the University of Salford, England. He is Associate Professor of computer science at the Iran University of Science and Technology. His research interests include reverse engineering and parallel and distributed computing.

Mohammad Hamzei received his B.Sc. degree from Razi University, Iran, and his M.Sc. degree from Iran University of Science and Technology, in 2007 and 2009, respectively, all in computer engineering. He is currently working towards his Ph.D. degree 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 and parallel programming models design and implementation.