Oil and Water can mix! Experiences with integrating Polyhedral and AST-based Transformations

Jun Shirako and Vivek Sarkar
Department of Computer Science, Rice University
Email: {shirako,vsarkar}@rice.edu

***Abstract*—The polyhedral model is an algebraic framework for affine program representations and transformations for enhancing locality and parallelism. Compared with traditional AST-based transformation frameworks, the polyhedral model can easily handle imperfectly nested loops and complex data dependences within and across loop nests in a unified framework. On the other hand, AST-based transformation frameworks for locality and parallelism have a long history that dates back to early vectorizing and parallelizing compilers. They can be used to efficiently perform a wide range of transformations including hierarchical parametric tiling, parallel reduction, scalar replacement and unroll-and-jam, and the implemented loop transformations are more compact (with smaller code size) than polyhedral frameworks. While many members of the polyhedral and AST-based transformation camps see the two frameworks as a mutually exclusive either-or choice, our experience has been that both frameworks can be integrated in a synergistic manner. In this paper, we present our early experiences with integrating polyhedral and AST-based transformations. Our preliminary experiments demonstrate the benefits of the proposed combined approach relative to Pluto, a pure polyhedral framework for locality and parallelism optimizations.**

I. INTRODUCTION

The polyhedral model [11] is an algebraic framework for affine program representations and transformations for enhancing locality and parallelism. Compared with traditional AST-based transformation frameworks [15], the polyhedral model can easily handle imperfectly nested loops and complex data dependences within and across loop nests in a unified framework. On the other hand, AST-based transformation frameworks for locality and parallelism have a long history that dates back to early vectorizing and parallelizing compilers. Compared with the polyhedral model, AST-based frameworks can easily provide cost models corresponding to the syntactic properties of the intermediate program representations. Though they lack the unified view of transformations present in polyhedral frameworks, AST-based frameworks efficiently support a wide range of transformations with proper cost models, including hierarchical parametric tiling, scalar replacement, unrolling, unroll-and-jam and vectorization. Another benefit of AST-based frameworks is that loop transformations can be implemented more compactly (with smaller code size) than polyhedral frameworks.

While many members of the polyhedral and AST-based transformation camps see the two frameworks as a mutually exclusive either-or choice, our experience has been that both frameworks can be integrated in a synergistic manner. In this paper, we present our early experiences at Rice University with integrating polyhedral and AST-based transformations in extensions to the ROSE compiler system [18]. These experiences build on our past work on analytical bounds for optimal tile size selection [20], [9], There are two goals derived from these experiences: one to bring the advantages of the polyhedral model - i.e., unified view of program representations/transformations - to the world of AST-based transformations, and another to enable cost models to be freely used in polyhedral frameworks. As an initial demonstration of integrating polyhedral and AST-based frameworks, we focus on a cost-based approach to locality optimization that identifies the best loop order for each statement, and also determines when loop fusion is profitable. Specially, our approach in this paper consists of the following three proposals.

1) Proposing a practical two-phase approach to integrate polyhedral and AST-based frameworks: former phase for locality optimization and latter phase for remained transformations.
2) Extending the polyhedral framework with a cost-based approach to locality optimization.
3) Providing a simple approach to extract data dependence information from polyhedral dependences as the form of dependence vectors (latter phase).

To be more specific, in this paper we first apply the polyhedral transformations with cost-based extensions so that loops with data locality can be legally interchanged and fused. This phase also includes index set shifting and loop reversal. In the latter phase, the fused loop nests are individually enabled for AST-based loop transformations with proper cost models, including skewing, hierarchical tiling, unrolling, unroll-and-jam, parallelization (doall, reduction and doacross), and vectorization. The dependence information from the polyhedral framework enhances the legality analysis for AST-based loop transformations, whereas these transformations can be performed more compactly and efficiently using an AST-based framework rather than the polyhedral framework. Our preliminary experiments demonstrate the benefits of the proposed combined approach relative to Pluto [8], a pure polyhedral framework for locality and parallelism optimizations.

The rest of the paper is organized as follows. Section II provides background on the polyhedral model and AST-based transformations. Section III introduces the cost-based analyses for best loop permutation order and loop fusion profitability. Section IV describes the overview of the proposed approach to integrate polyhedral and AST-based transformations, and Section V shows the details. Section VI presents the experimental results for the proposed integrating approach. Related work is discussed in Section VII, and we conclude in Section VIII.

II. BACKGROUND

A. Polyhedral Model

The polyhedral model [11] is an algebraic framework for affine program representations and transformations. The major strengths of polyhedral models over AST-based approaches include: 1) unified representation of perfectly and imperfectly nested loops and a large set of transformations that can be performed on them, and 2) mathematical approaches to express data dependences and to verify legality of transformations. These features provide a high degree of flexibility in specifying loop transformations in a unified framework. Benabderrahmane et al. summarized the advantages of the polyhedral model as follows [6]. *"To fight a common misunderstanding, the power of the polyhedral model is not to achieve exact data dependence analysis, but to implement compositions of complex transformations as a single algebraic operation, and to model these transformations in a convex optimization space"*.

On the other hand, not all loop transformations are straightforward to implement in the polyhedral framework. For instance, unrolling and unroll-and-jam have to alter the input polyhedral representation by adding new statements; loop strip-mining and loop tiling have to modify the iteration domains [14]. Furthermore, transformations such as software pipelining and statement-level doacross parallelization are hard to express in the polyhedral model. Another aspect of difficulty in the polyhedral model is due to the fact that the code generation phase is separated from the optimization (transformation) phase and is essentially a black box for the rest of the compiler. Therefore, syntactic properties of the generated code, e.g., which loop index has stride-1 array references, are complex to model in the optimization phase although such properties are important for cost analysis for loop transformations, e.g., vectorization [16].

We now briefly summarize some of the key aspects of the polyhedral model. Since our polyhedral transformation is influenced by the Pluto framework [8], we use the same terms/expressions as those used in Pluto. First, an integer polyhedron in n-dimensional space is defined as follows.

Definition 1 (Polyhedron): The set of all integer vectors $\vec{x} \in \mathbb{Z}^n$ such that $A\vec{x} + \vec{b} \geq \vec{0}$, where matrix $A \in \mathbb{Z}^{m \times n}$ and vector $\vec{b} \in \mathbb{Z}^m$, defines a (convex) integer polyhedron. A polytope is a bounded polyhedron with finite size.

Polyhedral representation of programs: Based on Definition 1, a program is expressed as a collection of polyhedra, which correspond to the statement set S in the program; each statement $S_i \in S$ is enclosed in one or more loop(s). Given a program, each dynamic instance[1] of a statement S_i is defined by an n-dimensional iteration vector $\vec{i}$ which contains values for the indices of the loops surrounding S_i, from outermost to innermost. Whenever the loop bounds are linear combinations of outer loop indices and program parameters represented by a vector $\vec{p}$ (typically, symbolic constants representing problem sizes), the set of dynamic iteration vectors corresponding to a statement define a polytope, $\mathcal{D}_{S_i}$, which is called *domain* of statement S_i.

Polyhedral Dependences: The dependence between statements is also represented by a polyhedron. The Polyhedral Dependence Graph (PDG) is a directed multi-graph $G = (V, E)$, where each vertex represents a statement, i.e., $V = S$, and each edge $e^{S_i \to S_j} \in E$ represents a dependence between dynamic instances of S_i and S_j[2]. For a dependence edge $e^{S_i \to S_j}$, let the source iteration of S_i be $\vec{s} \in \mathcal{D}_{\vec{s}}$ and target iteration of S_j be $\vec{t} \in \mathcal{D}_{\vec{t}}$, then these two instances access the same memory location and $\vec{s}$ is the last access before $\vec{t}$. This relation is expressed as a set of equations, i.e., $\vec{s} = h_e(\vec{t})$, which is also known as the *h-transformation* [12]. The dependence information corresponding to edge e is characterized by a polyhedron, $\mathcal{P}_e$, which is called the *dependence polyhedron*. The rank of the dependence polyhedron includes the sum of the dimensionalities of the source and target statements' polyhedra $\mathcal{D}_{\vec{s}}$ and $\mathcal{D}_{\vec{t}}$ with dimensions for program parameters as well. For $e \in E$, dependence polyhedra $\mathcal{P}$ is summarized as follows (for the case when no additional program parameters are included):

$$\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \to S_j}} \iff \vec{t} \in \mathcal{D}_{\vec{t}} \ depends \ on \ \vec{s} \in \mathcal{D}_{\vec{s}} \ via \ e^{S_i \to S_j}$$

Affine transformations: Finally, a d-dimensional affine transformation $\mathcal{T}_{S_i}$ for a statement S_i is expressed as follow. Let $M_{S_i} \in \mathbb{Z}^{d \times n}$, $\vec{i} \in \mathbb{Z}^n$ and $\vec{t}_{S_i} \in \mathbb{Z}^d$.

$$\mathcal{T}_{S_i}(\vec{i}) = M_{S_i}\vec{i} + \vec{t}_{S_i}$$

A legal affine transform must be a one-to-one mapping from the original domain $\mathcal{D}_{S_i}$ to transformed domain $\mathcal{T}(\mathcal{D}_{S_i})$, for all $S_i \in S$. Furthermore, all dependences must be satisfied, i.e., $\forall e^{S_i \to S_j} \in E : \mathcal{T}_{S_j}(\vec{t}) - \mathcal{T}_{S_i}(\vec{s}) \succ \vec{0}, \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \to S_j}}$. Note that $\vec{x} \succ \vec{0}$ means $\vec{x}$ is lexicographically positive and the above relation is equivalent to the traditional dependence condition: the dependence target must come after the dependence source[3].

B. AST-based Loop Transformations

The AST-based transformations can be considered as a sequence of these individual loop transformations applied to the Abstract Syntax Tree (AST) of the program region of interest. Several common loop transformations introduced by past work [15], [23] are summarized below and shown in Figure 6. Note that we assume parallelization and vectorization to be applied after locality optimizations.

1) **Loop Fusion** merges two or more loops into a single loop. This transformation can help improve data locality and coarse-grained parallelism.
2) **Loop Distribution** is the inverse of loop distribution. It divides the body of a loop into several loops for different parts of the loop body. This transformation can reduce the impact on cache and prefetch streams

[1] As in past work on polyhedral frameworks including Pluto, we will restrict our attention to dynamic instances of statements within a single instance of a procedure invocation. Formalizing dynamic instances of statements across both loop iterations and procedure invocations is beyond the scope of this paper.

[2] To be more specific, an edge e corresponds to a pair of particular references to read/write a memory location, e.g., array element, in S_i and S_j.

[3] $\mathcal{T}$ contains scalar dimensions to handle loop-independent dependences [8]

by decreasing loop body size. Also, some cases allow to cut loop-carried dependences, thereby exposing more doall parallelism.

3) **Loop Permutation (Interchange)** takes a multi-level perfectly nested loops and permutates the loop order. It can be used to improve data locality, coarse-grained parallelism, and vectorization opportunities.
4) **Loop Skewing** also takes a perfect loop nest and rearranges the loop iteration space and array subscripts, thereby changing loop dependences. It can be used to convert doacross parallelism into wavefront doall parallelism and increase permutability, i.e., help loop permutation, tiling and unroll-and-jam.
5) **Index Set Shifting** shifts the iteration space and array subscripts. This transformation is mainly used to align inter-loop dependences and increase the opportunities of loop fusion.
6) **Loop Reversal** inverts the direction of loop iteration. This transformation can also help other transformations such as fusion.
7) **Loop Peeling** separates the first/last iterations from the remained iterations and creates multiple loops, the prologue/epilogue loop and kernel loop. This transformation can simplify the loop body when it has different computations depending on the iteration, e.g., `if (i == 0) foo(); else bar();`.
8) **Loop Strip-Mining** is a loop transformation that replaces a single loop with two nested loops with smaller segments. This restructuring is an important preliminary step for transformations to improve locality and parallelism, such as tiling, unrolling, and unroll-and-jam.
9) **Loop Tiling** is the multi-nest version of loop strip-mining, which can be seen as the combination of multiple applications of strip mining and loop permutation. Especially, parametric loop tiling is an important technique because it can change the computation granularity of tiles at runtime so as to fit the data per tile within cache/TLB and control the computation/-communication trade-off when parallelized.
10) **Loop Unrolling** is to convert the innermost loop after strip mining into the equivalent sequence of the loop bodies. This transformation can enhance vectorization and reduce the overhead to iterate loop. Note that loop peeling to handle the last iteration is required when the original loop iteration count is not the multiple number of unrolling factor.
11) **Unroll-and-Jam** is the combination of loop strip-mining, permutation, and unrolling so as to enhance data reuse along with the unrolled loop. This transformation may also require loop peeling for the last iteration.

Although the phase-ordering problem among individual loop transformations is not trivial, each transformation can be applied to the intermediate AST with proper cost models that correspond to the syntactic property at each point.

There are AST-based frameworks that apply a sequence of transformations to a single loop nest [19], [3], [19], [4], [21] including loop permutation, skewing, strip-mining, tiling, unrolling, unroll-and-jam vectorization, and doacross-parallelization. However, a big challenge for AST-based frameworks is optimizations for complex loop structures including imperfectly nested loops, e.g., inter-loop data localization, in a unified manner as with the polyhedral framework.

III. Cost-based Analysis via DL Model

As an initial demonstration of integrating polyhedral and AST-based frameworks, we focus on a cost-based approach to locality optimization that identifies the best loop order for each statement, and also determines when loop fusion is profitable. This section introduces a cost-based analysis for loop permutation order and loop fusion using the DL model [19], [20], which gives the number of distinct cache lines accessed in a loop nest.

A. DL Model

The DL (Distinct Lines) model, which forms the basis for our cost-based approaches, was designed to estimate the number of distinct cache lines accessed in a loop-nest [13], [19]. Consider a reference to a contiguously allocated m-dimensional array, A, enclosed in n perfectly nested loops, with index variables $i_1, \cdots, i_n$:

$$A\left(f_1(i_1,\cdots,i_n),\cdots,f_m(i_1,\cdots,i_n)\right) \quad (Fortran)$$

$$A[f_m(i_1,\cdots,i_n)]\cdots[f_1(i_1,\cdots,i_n)] \quad (C),$$

where $f_j(i_1, \cdots, i_n)$ is an affine function. An exact analysis to compute DL is only performed for array references in which all coefficients are compile-time constants (i.e., for affine references). An upper bound for the number of distinct lines accessed by a single array reference [13] with one-dimensional subscript expression $f(i_1, \cdots, i_n)$ is

$$DL(f) \leq \min\left(\frac{(f^{hi}-f^{lo})}{g}+1, \left\lceil\frac{(f^{hi}-f^{lo})}{L}\right\rceil+1\right),$$

where g is the greatest common divisor of the coefficients of the enclosing loop indices in f, and L is the cache line size in units of array element size; f^{hi} and f^{lo} are the maximum and minimum values of the subscript expression f across the entire loop nest. In practice, the relative error of this estimation is small when, as is usually the case, the range $(f^{hi} - f^{lo})$ is much larger than the values of the individual coefficients of f. For a multidimensional array reference $A\left(f_1, \cdots, f_m\right)$, the upper bound estimate [13] is as follows[4].

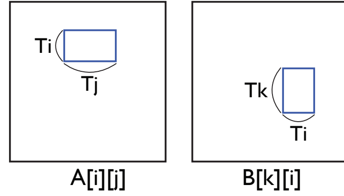
$$DL(f_1,\cdots,f_m) = DL(f_1) \times \prod_{j=2}^{m}\left(\frac{(f_j^{hi}-f_j^{lo})}{g_j}+1\right)$$

Extensions of this model to account for multiple accesses to the same array in a loop nest have also been developed [13], [19]. These DL definitions for a loop nest are also applicable to a loop nest after loop tiling is performed. Given a tiled loop nest whose loop boundaries are expressed using tile sizes, $t_1, t_2, \cdots, t_n$, the DL definition is a symbolic function of tile sizes denoted by $DL(t_1, t_2, \cdots, t_n)$ [19].

In the following discussion, we use the DL expression of each individual statement so as to enable DL-based analyses

[4] This is based on a heuristic assumption that the first dimension of the array has at least L elements. The more general case without this assumption is also discussed in [19].

```
for ti = 0, N-1, Ti
 for tj = 0, M-1, Tj
  for tk = 0, K-1, Tk
   for i = ti, ti+Ti-1
    for j = tj, tj+Tj-1
     for k = tk, tk+Tk-1
      A[i][j] += B[k][i];
```



DL = DL$_A$ + DL$_B$
= Ti * ⌈Tj / L⌉ + Tk * ⌈Ti / L⌉

mem_cost = C$_{line}$ * DL / (Ti * Tj * Tk)
= C$_{line}$ * (⌈Tj / L⌉ / (Ti * Tk) + ⌈Ti / L⌉ / (Ti * Tj))

- Selects Ti, Tj, and Tk such that all data of a tile fits within a specific cache
- L : cache line size (# array elements per line)
- C$_{line}$: memory cost per cache line

Fig. 1: Example for DL and memory cost

in the polyhedral model. This is a natural extension of our previous work [19]. Note that the definition of DL and memory cost can also be applied to any level of cache or TLB by selecting its cache line size or page size as L.

B. Memory Cost

First, we assume loop tiling so as to enhance data reuse by fitting all the data per tile within a particular cache/TLB and hence the DL expression is a function of tile sizes $t_1, t_2, \cdots, t_n$ as described in Section III-A. However, all the analyses in this section are not specific to loops that need tiling since non-tiled loop can simply be modeled as "tile size = loop iteration size". Based on this DL expression, a per-iteration memory cost of a statement is defined as follow.

$$mem_cost(t_1, t_2, \cdots, t_n) = \frac{C_{line} \times DL(t_1, \cdots, t_n)}{t_1 \times t_2 \times \cdots \times t_n}$$

This assumes that the all distinct lines of a tile are kept on a specific cache, and C_{line} represents the memory cost (cache miss penalty) per cache line. Therefore, $C_{line} \times DL$ gives the total cost to bring all the data within a tile to the particular cache memory (Figure 1). Our analyses for loop permutation order and fusion profitability is based on this mem_cost as shown below. Therefore, the results can be affected by architectural parameters. In the experimental results, we simply use the DL expression based on the level-1 cache of the target platform.

C. Best Permutation Order Analysis

The partial derivative of mem_cost with respect to tile size t_i - i.e., $\delta mem_cost/\delta t_i$ - represents the variation rate of memory cost when tile size t_i is increased. For instance, $\delta mem_cost/\delta t_i < 0$ indicates that increasing t_i causes a decrease in memory cost. We use $\delta mem_cost/\delta t_i$ as the priority for loop permutation order because placing the loop with the most negative value at the innermost position could yield the largest benefit on data locality. As a heuristic approach [19], we compute $\delta mem_cost/\delta t_i$ at a default point, e.g., $(t_1 = 1, t_2 = 1, \cdots, t_n = 1)$, for all loops of $1 \leq i \leq n$ and the decreasing order is used as the most profitable loop order.

D. Loop Fusion Profitability Analysis

A straightforward approach to determine whether loop fusion is profitable is to compare the DL memory costs before and after loop fusion. For instance, given two loop nests that share arrays with same access patterns (Figure 2a), the memory cost of fused loop nest should be smaller than the total memory cost of two individual loop nests because the shared array, e.g., `A[i][j]` of `S2`, should already be in the cache/TLB at the second access and hence the cost for the second array reference can be ignored. In general cases, however, comparing the memory costs before/after fusion is not as simple as the analysis for loop permutation order because memory cost mem_cost is a function of tile sizes and it has to find the tile size that minimizes mem_cost under the constraint that the all data of a tile of the fused loop fits within a specific cache/TLB [13], [19], [20]. Note that the above boundary constraint generally becomes stricter after loop fusion while larger tile size gives smaller memory cost when the loop nest has data locality [13], [19], [20]. This boundary constraint on tile sizes is analytically defined based on the DL expression and architectural parameters [20]. Therefore, it is possible to define the functions for mem_cost and tile size boundaries before and after loop fusion and compare the minimum memory costs although the results could be affected by various platform parameters [20].

Alternatively, we can estimate the fusion profitability simply based on the array access patterns of shared arrays. To be more specific, we will judge the fusion of two loops at the same nest-level is profitable if there is at least one array which the two loops access with the same pattern. This approach also has a strength that it can make the decision of loop fusion for each nest-level in a stepwise manner. According to the definitions in the DL model, an m-dimensional array reference $A[f_m][f_{m-1}]\cdots[f_1]$ contains array subscripts of affine functions.

$$\begin{pmatrix} f_1 \\ f_2 \\ \cdots \\ f_m \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ & & \cdots & \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ \cdots \\ i_n \end{pmatrix} + \begin{pmatrix} c_{10} \\ c_{20} \\ \cdots \\ c_{m0} \end{pmatrix}$$

Note that permutation of loop-i_j and loop-i_k is equivalent to exchanging indices i_j and i_k in the index vector $\vec{i}$ and corresponding columns c_{*j} and c_{*k} in the coefficient matrix C.

Loop fusion policy: Let us consider two n-th nested perfect loop nests, which respectively have statements S1 and S2 as their loop body, each of which contains an access to an m-dimensional array A. The coefficient matrices for the two references to array A are represented as $C^{S1}_{m,n}$ and $C^{S2}_{m,n}$. When these loop nests are partially fused until the $(k-1)$-th nest-level, our heuristic decides to fuse the loops at the k-th nest level if the columns c^{S1}_{*l} and c^{S2}_{*l} are identical for $1 \leq l \leq k$. Figure 2b shows an example that has two matrix-multiplications after loop permutation with the best order. Our approach fuses only the outermost i-loops because columns c^{S1}_{*1} and c^{S2}_{*1} are identical but others are not.

The profitability of loop fusion will be affected by various aspects such as number of registers and prefetch streams. The

```
        for i = 0, N-1
          for j = 0, N-1
S1:         A[i][j] = foo(i,j);
        for i = 0, N-1
          for j = 0, N-1
S2:         sum += A[i][j];

        // After loop fusion
        for i = 0, N-1
          for j = 0, N-1
S1:         A[i][j] = foo(i,j);
S2:         sum += A[i][j];
```

(a) Summation

```
        for i = 0, N-1
          for k = 0, N-1
            for j = 0, N-1
S1:           A[i][j] += B[i][k]
                       * C[k][j];

        for i = 0, N-1
          for k = 0, N-1
            for j = 0, N-1
S2:           D[i][j] += A[i][k]
                       * E[k][j];
```

$$\begin{pmatrix} f^{A}_{1} \\ f^{A}_{2} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ k \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} f^{A}_{1} \\ f^{A}_{2} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ k \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

(b) 2 matrix-multiplications with loop interchange

Fig. 2: Example for loop fusion

extension of the fusion profitability analysis to support/integrate other analytical models is an important future work.

IV. Two-phase Approach to Integration of Polyhedral and AST-based Transformations

As described in Section II-A, the major strength of the polyhedral model over AST-based approaches is the ability to handle complex transformations on any loop structure for perfectly and imperfectly nested loops. On the other hand, Section II-B shows that the AST-based transformations can easily analyze and transform intermediate state of program structures at the AST level, which are not easily available in the polyhedral optimizations. The proposed integration approach is based on two goals: one to bring the advantages of the polyhedral model to the world of AST-based transformations, and another to enable cost models to be used in polyhedral frameworks. This section is an introduction to the proposed two-phase approach to integrate polyhedral and AST-based transformations. We start with the overview of the two-phase approach in Section IV-A and Section IV-B discusses the design decision.

A. Overview

As discussed in Section II-B, a big challenge for AST-based frameworks is to handle the inter-loop data locality among different loop nests though several frameworks have been proposed to optimize a single loop nest, i.e., perfectly nested loops. This challenge could be addressed by building a loop fusion framework that also supports other loop transformations related to fusion and merges different loop nests with data locality into a single loop nest. Such a framework should be implemented with the polyhedral model because of its ability to model transformations of multiple loop nests in a unified manner.

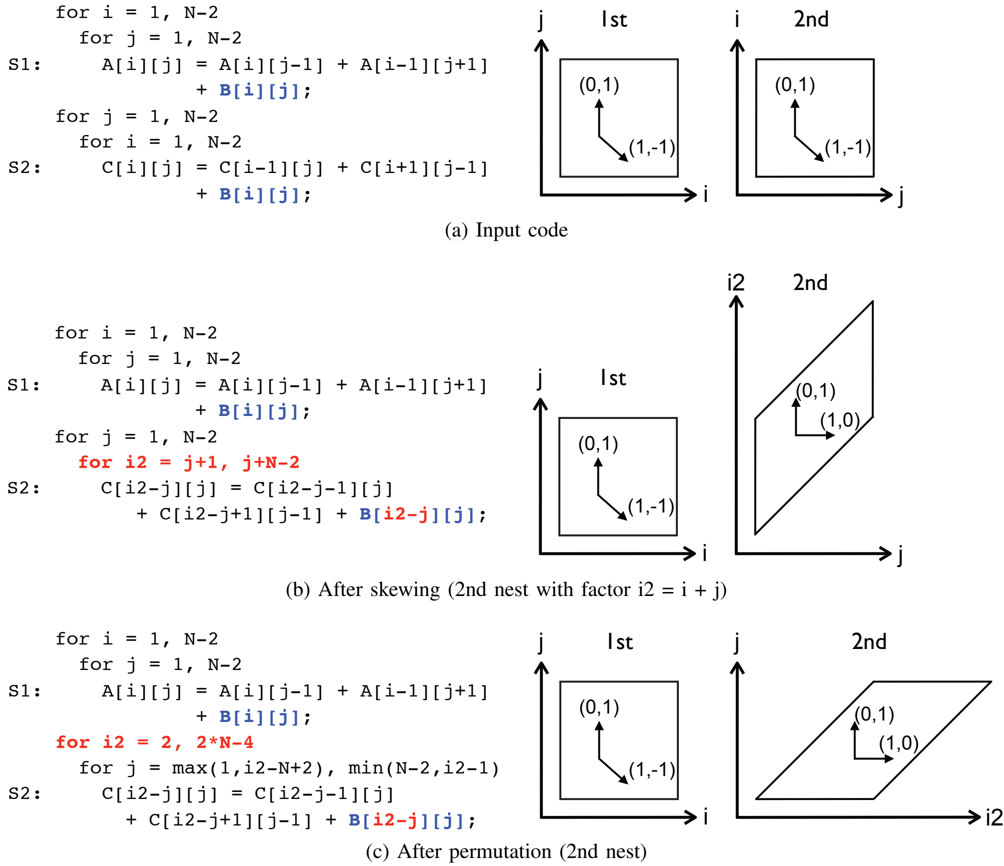
A natural question that arises is how to decide which loops to fuse. As discussed in Section III, DL-based fusion profitability analysis can be used to answer that question. Further, the possible combination of loop fusion is directly affected by loop permutation order, while the loop order also has significant impact on various aspects of performance such as data locality, vectorization efficiency, and parallelism. The partial derivative analysis of the DL memory cost is one of the metrics that characterize these aspects and estimate the most profitable permutation order. In this paper, the former phase of the proposed two-phase integration approach is the polyhedral transformation extended with the DL-based analyses for loop fusion and permutations. Section V-A describes how the DL-based analyses utilized in the polyhedral model.

The output of the former phase - i.e., the input to the latter phase - is a set of fused loop nests; another sequence of AST loop transformations can be individually applied to each fused loop nest. In this paper, we focus on the AST-based approach to implement the latter phase because of its flexibility to apply the sequence of individual transformations to a single loop nest with proper syntactic properties, and compact and efficient code generation. Section V-B addresses the latter phase by the AST-based transformations.

B. Design Decision for Two-phase Integration Approach

In this section, we discuss which individual loop transformations should be supported in the polyhedral phase, *phase-1*, and which to be in the AST phase, *phase-2*. As discussed in Section IV-A, loop fusion and permutation must be handled close together and hence they should be in phase-1. In addition to these transformations, index set shifting and loop reversal also increase the opportunities of loop fusion as briefly introduced in Section II-B. Further, loop skewing changes the loop dependence so as to increase the permutability - i.e., applicability of loop permutation. However, we do not include skewing in phase-1 because it has adverse effects for loop fusion in many cases that arise in practice. First, let us review the effect of loop skewing. When we consider the iteration space of a perfect loop nest, loop skewing can be viewed as an affine map of the iteration space; the transformed loop nest has different loop boundaries, array reference patterns, and loop dependences from the original loop nest. The by-product of changing the iteration space and array reference patterns could affect on the inter-loop data locality, thereby reducing the benefit of loop fusion in practical cases.

Figure 3a is a typical example where loop skewing is necessary before loop permutation and fusion. Both loop nests access to array `B` with the reference pattern of `[i][j]`; applying loop permutation to either nest and then fusion should improve the inter-loop data locality. However, both nests contain the dependence distance vector $(1, -1)$ and permutation is impossible because it results in an illegal



(a) Input code

(b) After skewing (2nd nest with factor i2 = i + j)

(c) After permutation (2nd nest)

Fig. 3: Affect of loop skewing on inter-loop data locality

dependence, $(-1, 1)$. In Figure 3b, loop skewing is applied to the first loop nest and the dependence vector $(1, -1)$ is converted into $(1, 0)$, which allows legal loop permutation, and Figure 3c shows the code and iteration spaces after the permutation of the second loop nest. Here, we can see the skewed loop index `i2` is nothing like the original index `i` because of the converted iteration space and array reference, `[i2-j][j]`. Loop fusion is now possible, but the array references to `B[i][j]` and `B[i2-j][j]` no longer access the same/neighboring elements of array `B`. Further, the fused loop nest has to manage the complicated loop boundaries and statements within the loop body. Finally, the information for profitable loop permutation order no longer makes sense after loop skewing, because it is for the original loop index `i`, not for the new index `i2`.

Based on the above observation, we define phase-1 to be a polyhedral framework and supporting transformations of permutation, index set shifting and reversal. Loop skewing is included in phase-2 so as to increase permutability and help related transformations e.g., tiling and unroll-and-jam. In this paper, phase-2 is defined as the AST-based framework to handle any loop transformation that can be applied to a single loop nest, including loop skewing, tiling, unrolling, unroll-and-jam, parallelization (doall, reduction and doacross), and vectorization.

V. Details of Two-phase Integration Approach

This section describes the details of the proposed two-phase approach. Due to the page size limit, we focus on the detailed algorithms of phase-1 in Section V-A while Section V-B shows the key ideas of phase-2.

A. Phase-1: Polyhedral Transformations with DL Analysis

$$\mathcal{T}(\vec{i}) = \begin{pmatrix} \upsilon_1(\vec{i}) \\ \phi_1(\vec{i}) \\ \upsilon_2(\vec{i}) \\ \phi_2(\vec{i}) \\ \cdots \\ \upsilon_n(\vec{i}) \\ \phi_n(\vec{i}) \\ \upsilon_{n+1}(\vec{i}) \end{pmatrix} = \begin{pmatrix} 0 \\ sign_1 \; i_{\pi(1)} \\ 0 \\ sign_2 \; i_{\pi(2)} \\ \cdots \\ 0 \\ sign_n \; i_{\pi(n)} \\ 0 \end{pmatrix} + \begin{pmatrix} b_1 \\ c_1 \\ b_2 \\ c_2 \\ \cdots \\ b_n \\ c_n \\ b_{n+1} \end{pmatrix}$$

Given a statement enclosed in n nested loops with index variables $i_1, \cdots, i_n$, the general expression of statement-wise affine transform $\mathcal{T}$ for the proposed approach is shown above. Note that we explicitly show the scalar dimensions of $\mathcal{T}$ with the symbol of υ, while ϕ is used for regular loop dimensions [8]. The value of $sign_k$ is either 1 for positive sign or -1 for negative sign; π is the permutation of n elements such that

$\pi(k)$ represents the element after permutation[5]. We can see the following relations between the above affine expression and the loop restructurings supported in the phase-1 transformation.

- **Loop fusion** $\leftrightarrow b_k$**:** If two statements S_i and S_j are fused until k-th nest-level, then $b_l^{S_i} = b_l^{S_j}$ for $(1 \leq l \leq k)$.
- **Loop permutation** $\leftrightarrow \pi(k)$**:** The permutation π directly represents the loop permutation order - i.e., the k-th loop index after permutation is $i_{\pi(k)}$.
- **Index set shifting** $\leftrightarrow c_k$**:** Index set shifting factor is equivalent to c_k.
- **Loop reversal** $\leftrightarrow sign_k$**:** The application of loop reversal is represented as a negative sign of $sign_k$.

The details of the proposed polyhedral loop fusion that determines suitable values of b_k, $\pi(k)$, c_k and $sign_k$ are shown in Algorithms 1, 2 and 3. Algorithm 1, which is the entry point to the overall transformations, starts from the top level $k = 1$ with all statements within the program region of interest and corresponding dependence polyhedra. As described in Section III-C, the partial derivative of memory cost gives the most profitable permutation order of each statement S_i and the order is stored into $Best^{S_i}$. Also, when the loop orders of two statements S_i and S_j are fixed until k-th nest-level as $\{\pi^{S_i}(1), \pi^{S_i}(2), \ldots, \pi^{S_i}(k)\}$ and $\{\pi^{S_j}(1), \pi^{S_j}(2), \ldots, \pi^{S_j}(k)\}$ respectively and they are fused until nest-level k-1, the profitability of loop fusion at nest-level k is analyzable based on the coefficients of array references in S_i and S_j as discussed in Section III-D.

Algorithm 1 first computes Strongly Connected Components (SCC) of statement set S at nest-level k based on dependence edges in E (line 3). Note that the statements in a SCC, which is denoted by Scc_a in the algorithms, must be fused at level k to satisfy the dependences within Scc_a. Algorithm 2 is applied to each Scc_a so as to determine $\pi(k)$, the loop permutation at level k (line 7). If Scc_a contains only a statement S_i which is not enclosed in a loop at level k, i.e., the dimensionality of S_i is $k-1$, the affine transform $\phi_k^{S_i}$ is undefined and no permutation nor index set shifting at level k is required. Such statements are stored into $SnglStmSet$ (line 10) and split from the set of regular SCCs, $SccSet$ (line 11). Based on the loop permutation order until level k, Algorithm 3 determines which SCCs should be fused at level k (line 13). Further, the position of each statement of $SnglStmSet$ at level k is determined to satisfy its dependences (line 15). These decisions for lines 13 and 15 are equivalent to the selection of b_k. Above Algorithms 2 and 3 also provide legality constraints on $sign_k$ and c_k, which represent the application of loop reversal and selection of index set shifting factor. The current version of algorithms simply selects $sign_k$ as 1 when it does not violate dependences, and c_k as some value that is legal and to minimize dependence distance (line 16). Note that the selected index set shifting factor c_k in the phase-1 transformation is a tentative value because index set shifting affects on loop skewing and should be also applied in the phase-2 transformation. A dependence edge e is removed from E if the above solutions for υ_k and ϕ_k guarantee the legality constraint for e - i.e., $\upsilon_k(\vec{t}) - \upsilon_k(\vec{s}) > 0$ or $\upsilon_k(\vec{t}) - \upsilon_k(\vec{s}) = 0 \wedge \phi_k(\vec{t}) - \phi_k(\vec{s}) > 0$, $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_e$ (line17). Finally, Algorithm 1 is recursively applied to the statements in $Group_x$, which are fused into a loop at level k (line 21).

[5]If $\{1, 2, 3\}$ is permutated as $\{2, 3, 1\}$, then $\pi(1) = 2$, $\pi(2) = 3$, and $\pi(3) = 1$.

Algorithm 1: Affine transformation for phase-1

Input : k : current nest-level (top level is 1),
S : set of statements S_i fused until level k-1,
$Best^{S_i}$, $S_i \in S$: best permutation order for S_i,
E : set of dependence edges $e^{S_i \rightarrow S_j}$,
$\mathcal{P}_e$, $e \in E$: dependence polyhedron for e

Output: Affine transforms $\upsilon_l(\vec{i}) = b_l$ and
$\phi_l(\vec{i}) = sign_l \ i_{\pi(l)} + c_l$ for $k \leq l \leq n$

```
1  begin
2      SnglStmSet := ∅
3      SccSet := computes SCCs of S at level k via E
4      // Intra-SCC transformation to determine π(k)
5      for each Scc_a ∈ SccSet do
6          if n^{Scc_a} ≥ k then
7              Applies Algorithm 2 to Scc_a
8          else
9              // Scc_a contains only a non-loop statement
10             SnglStmSet := SnglStmSet ∪ Scc_a
11             Remove Scc_a from SccSet
12     // Inter-SCC transformation to determine b_k
13     Applies Algorithm 3 to SccSet
14     for each S_i ∈ SnglStmSet do
15         Determines b_k^{S_i} to satisfy dependences for S_i
16     Determines sign_k and c_k to satisfy all constraints
       from Algorithms 2 and 3
17     Removes dependence e from E if e is guaranteed
       by the above solutions
18     // GroupSet is defined by Algorithm 3
19     for each Group_x ∈ GroupSet do
20         // Statements in Group_x were fused at level k
21         Applies Algorithm 1 to Group_x for k' := k + 1
22 end
```

Algorithm 2 shows the details to determine the loop permutation at level-k, $\pi(k)$, for the statements in a given SCC, Scc_a. Based on the most permutable loop order $Best^{S_i}$ and loop fusion profitability analysis described in Section III-D, v-th combination of loops that can be located at level k is selected (line 7). Note that a combination with smaller v contains more profitable loops according to $Best^{S_i}$. As described in Section II-A, a legal affine transform must not violate any dependence at level k and the legality constraint for dependence polyhedron $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}}$, $e \in E$ is:

$$\phi_k^{S_j}(\vec{t}) - \phi_k^{S_i}(\vec{s}) =$$
$$sign_k^{S_j} t_{\pi^{S_j}(k)} - sign_k^{S_i} s_{\pi^{S_i}(k)} + c_k^{S_j} - c_k^{S_i} \geq 0$$

The inner loop from line 8 to line 11 collects the above legality constraints for all dependences, and the outer loop from line 3 to line 12 repeats this process until it founds the legal

Algorithm 2: Loop permutation and fusion within a SCC

Input : k : current nest-level,
Scc_a : set of statements S_i in a SCC,
$Best^{S_i}$, $S_i \in S$: best permutation order for S_i,
E : set of dependence edges $e^{S_i \rightarrow S_j}$,
$\mathcal{P}_e$, $e \in E$: dependence polyhedron for e

Output: Part of affine transforms $\upsilon_k(\vec{i}) = b_k$ and $\phi_k(\vec{i}) = sign_k \ i_{\pi(k)} + c_k$

1 **begin**
2 v := 0 // version for combination of loops at level k
3 **repeat**
4 $C := \emptyset$
5 $v := v + 1$
6 // $Scc_a(i)$ is the i-th statement in Scc_a
7 $\{\pi^{Scc_a(1)}(k), \pi^{Scc_a(2)}(k), \ldots\}$:= gets v-th combination of loops at level k via $Best^{S_i}$
8 **for** each $e^{S_i \rightarrow S_j} \in E^{Scc_a}$ **do**
9 // Legality Constraint for $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}}$
10 $C_l := "\phi_k^{S_j}(\vec{t}) - \phi_k^{S_i}(\vec{s}) \geq 0"$
11 $C := C \cup C_l$
12 **until** solutions of $sign_k$ and c_k for C exist ;
13 **for** each $S_i \in Scc_a$ **do**
14 // All statements in Scc_a are fused
15 Replaces $b_k^{S_i}$ by $b_k^{Scc_a(1)}$ in $\upsilon_k^{S_i}$
16 Removes $\pi^{S_i}(k)$ from $Best^{S_i}$
17 **end**

loop combination of $\pi(k)$ whose legality constraints C have solutions of $sign_k$ and c_k for all statements in Scc_a. Finally, b_k for all statements in Scc_a are forced to be identical because of loop fusion (line 15) and the selected loop $\pi(k)$ is removed from the set of best permutation order to avoid locating a loop at different nest-levels multiple times (line 16).

Algorithm 3 shows the details to determine the loop fusion among SCCs at level k. Based on the loop permutation orders selected by Algorithm 2, the information regarding which loops are *profitable* to be fused are summarized and stored in a 2-D array, $profit^{S_i,S_j}$ (line 3). We employed the following heuristic loop fusion algorithm that corresponds to the loop from line 6 to line 26. First, it picks up the first node, Scc_a, from the $SccSet$ and stores into $Group_x$, which is to contain all SCCs to be fused with each other (lines 7–8). The loop from line 9 to line 22 iterates over $SccSet$ multiple times so as to find all $Scc_b \in SccSet$ for legal and profitable loop fusion, where the criterion of profitability is $profit^{S_i,S_j}$ (line 12) and legality criteria consist of the following two check points. First, the loop fusion is only applicable to Scc_b that is direct predecessor/successor or has no dependence for the SCCs in $Group_x$ (line 11). Second, the same legality constraints for affine transforms as described in Algorithm 2 must be satisfied (lines 13–18). Scc_b that satisfies the above criteria is moved from $SccSet$ to $Group_x$ (line 19 and 20) and the corresponding legality constraints on $sign_k$ and c_k are collected (line 21). Finally, b_k for all statements in $Group_x$ are forced to be identical (line 24). The above process is repeated until $SccSet$ becomes empty (line 26), and all $Group_x$ is collected in $GroupSet$ (line 25).

Algorithm 3: Loop fusion among SCCs

Input : k : current nest-level,
S : set of statements S_i,
$SccSet$: set of SCCs,
E : set of dependence edges e,
$\mathcal{P}_e$, $e \in E$: dependence polyhedron for e

Output: Part of affine transforms $\upsilon_k(\vec{i}) = b_k$ and $\phi_k(\vec{i}) = sign_k \ i_{\pi(k)} + c_k$

1 **begin**
2 **for** each $\{S_i, S_j\} \in S \times S$ **do**
3 $profit^{S_i,S_j}$:= fusion profitability for S_i with π^{S_i} and S_j with π^{S_j}
4 $C := \emptyset$
5 $GroupSet := \emptyset$
6 **repeat**
7 Scc_a := pop the first SCC in $SccSet$
8 $Group_x$:= { Scc_a } // Set of SCCs to be fused
9 **repeat**
10 **for** each $Scc_b \in SccSet$ **do**
11 **if** Scc_b is not indirect
12 predecessor/successor of $Group_x \wedge \exists\{S_i, S_j\} \in Group_x \times Scc_b : profit^{S_i,S_j}$ **then**
13 $C2 := \emptyset$
14 **for** each $e^{S_i \rightarrow S_j} \in E^{Group_x \leftrightarrow Scc_b}$ **do**
15 // Constraint for $\langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}}$
16 $C_l := "\phi_k^{S_j}(\vec{t}) - \phi_k^{S_i}(\vec{s}) \geq 0"$
17 $C2 := C2 \cup C_l$
18 **if** solutions for $C2$ exist **then**
19 Removes Scc_b from $SccSet$
20 $Group_x := Group_x \cup \{Scc_b\}$
21 $C := C \cup C2$
22 **until** $Group_x$ is unchanged during the iteration ;
23 **for** each $S_i \in Group_x$ **do**
24 Replaces $b_k^{S_i}$ by $b_k^{Scc_a(1)}$ in $\upsilon_k^{S_i}$
25 $GroupSet := GroupSet \cup \{Group_x\}$
26 **until** $SccSet = \emptyset$;
27 **end**

B. Phase-2: AST-based Transformations with Polyhedral-derived Dependence

Theoretically, any AST-based transformation framework that is applicable to a single loop nest could work as the phase-2 of the proposed two-phase approach. In this section, we do not discuss the detailed algorithms for such frameworks. Instead, we focus on an approach that extracts data dependence information from polyhedral dependences as the form of dependence vectors, thereby enhancing AST-based loop transformations based on dependence vectors.

Extracting dependence vectors from polyhedral dependences: Given two statements S_i and S_j fused until nest-level n - i.e., $\upsilon_k^{S_i} = \upsilon_k^{S_j}, 1 \leq k \leq n$, the dependence distance vector for $e^{S_i \rightarrow S_j} \in E$ is represented using the notion of $\delta_e = \phi(\vec{t}) - \phi(\vec{s})$ as follow [8].

```
for i = 0, N-1
  for j = 0, N-1
    for k = 0, N-1
      A[i][j] += B[i][k] * C[k][j];

for l = 0, N-1
  for m = 0, N-1
    for n = 0, N-1
      D[l][m] += E[l][n] * A[n][m];
```

(a) 2mm type-1: Input

```
for i = 0, N-1
  for j = 0, N-1
    for k = 0, N-1
      A[i][j] += B[i][k] * C[k][j];

for l = 0, N-1
  for m = 0, N-1
    for n = 0, N-1
      D[l][m] += A[l][n] * E[n][m];
```

(b) 2mm type-2: Input

```
for i1 = 0, N-1
  for i2 = 0, N-1 {
    for i3 = 0, N-1
      A[i1][i2] += B[i1][i3] * C[i3][i2];

    for i3 = 0, N-1
      D[i3][i2] += E[i3][i1] * A[i1][i2];
  }
```

(c) 2mm type-1: Pluto

```
for i1 = 0, N-1
  for i2 = 0, N-1
    for i3 = 0, N-1
      A[i1][i3] += B[i1][i2] * C[i2][i3];

for i1 = 0, N-1
  for i2 = 0, N-1
    for i3 = 0, N-1
      D[i1][i3] += E[i1][i2] * A[i2][i3];
```

(d) 2mm type-1: Phase-1

```
for i1 = 0, N-1
  for i2 = 0, N-1 {
    for i3 = 0, N-1
      A[i1][i2] += B[i1][i3] * C[i3][i2];

    for i3 = 0, N-1
      D[i1][i3] += A[i1][i2] * E[i2][i3];
  }
```

(e) 2mm type-2: Pluto

```
for i1 = 0, N-1 {
  for i2 = 0, N-1
    for i3 = 0, N-1
      A[i1][i3] += B[i1][i2] * C[i2][i3];

  for i2 = 0, N-1
    for i3 = 0, N-1
      D[i1][i3] += A[i1][i2] * E[i2][i3];
}
```

(f) 2mm type-2: Phase-1

Fig. 4: Input and transformed codes for 2mm

$$\Delta_e = \begin{pmatrix} \delta_{e1} \\ \delta_{e2} \\ \cdots \\ \delta_{en} \end{pmatrix} = \begin{pmatrix} \phi_1^{S_j}(\vec{t}) - \phi_1^{S_i}(\vec{s}) \\ \phi_2^{S_j}(\vec{t}) - \phi_2^{S_i}(\vec{s}) \\ \cdots \\ \phi_n^{S_j}(\vec{t}) - \phi_n^{S_i}(\vec{s}) \end{pmatrix}, \langle \vec{s}, \vec{t} \rangle \in \mathcal{P}_{e^{S_i \to S_j}}$$

As shown in Section II-A, $\vec{s}$ is the last access before $\vec{t}$ accessing the same memory location and the relation between $\vec{s}$ and $\vec{t}$ is expressed by the h-transformation, $\vec{s} = h_e(\vec{t})$. Based on our loop fusion policy in Section III-D, the fused statements most likely have similar access patterns to the arrays with data dependences. Therefore, the result of $\phi_i^{S_j}(\vec{t}) - \phi_i^{S_i}(\vec{s}) = \phi_i^{S_j}(\vec{t}) - \phi_i^{S_i}(h_e(\vec{t}))$ tends to be a constant value. When all the dimensions are constant, Δ_e is directly used as the constant dependence distance vector for such $e \in E$.

We will address the cases where Δ_e contains some elements of $\vec{t}$ - i.e., loop indices - in our future work. We are thinking our past work on analytical tile size bounds based on reuse distance analysis could be one of key ideas to handle this issue. Although this approach would not cover the polyhedral dependence completely, we believe many practical cases could be addressed via that approach.

VI. Experimental Results

This section shows our preliminary experiments to demonstrate the proposed two-phase approach to integrate polyhedral and AST-based transformations. We use two benchmarks from PolyBench [1], 2mm and 2d-jacobi. 2mm contains two matrix multiplication kernels, where the first kernel defines one array that is used in the second kernel. For the experiments, we provided two versions, 2mm type-1 and 2mm type-2, where type-1 is equivalent to the original in PolyBench while type-2 has different array access patterns in the second kernel (Figures 4a and 4b). 2d-jacobi is a 5-point 2-dimensional stencil computation kernel. For these three benchmarks, we compare the proposed two-phase approach with Pluto [8], a pure polyhedral framework for locality and parallelism optimizations. The performance numbers are corrected on Intel Xeon E7330 2.4GHz quad core processor.

A. 2mm: Two Matrix Multiplication kernels

Figure 4 shows the input and output codes via Pluto and the two-phase approach for 2mm type-1 and type-2. To increase the readability, figures for the Pluto transformation show the codes before applying loop tiling, and figures for the two-phase approach show the codes after the phase-1 transformation (before applying phase-2). As shown in Figures 4a and 4b, the first and second loop nests share array A in both versions. The cost function to be minimized in the Pluto framework is based on dependence distances [8] and generally loops tend to

```
for t = 0, N-1 {
  for i = 1, M-1
    for j = 1, M-1
      B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1] + A[i+1][j] + A[i-1][j]);

  for i = 1, M-1
    for j = 1, M-1
      A[i][j] = B[i][j];
}
```

(a) 2d-jacobi: Input

```
for i1 = 0, N-1
  for i2 = 2*i1 + 1, 2*i1 + M
    for i3 = 2*i1 + 1, 2*i1 + M {
      i = i2 - 2*i1;
      j = i3 - 2*i1;
      if (i >= 2 && j >= 2)
        A[i-1][j-1] = B[i-1][j-1];
      if (i <= M-1 && j <= M-1)
        B[i][j] = 0.2 * (A[i][j] + A[i][j-1]
         + A[i][j+1] + A[i+1][j] + A[i-1][j]);
    }
```

(b) 2d-jacobi: Pluto

```
for i1 = 0, N-1
  for i2 = 1, M
    for i3 = 1, M {
      i = i2;
      j = i3;
      if (i >= 2 && j >=2)
        A[i-1][j-1] = B[i-1][j-1];
      if (i <= M-1 && j <= M-1)
        B[i][j] = 0.2 * (A[i][j] + A[i][j-1]
         + A[i][j+1] + A[i+1][j] + A[i-1][j]);
    }
```

(c) 2d-jacobi: Phase-1

Fig. 5: Input and transformed codes for 2d-jacobi

be fused as long as it is legal to do so. As a consequence, the outermost and middle loops are fused by Pluto in both versions as shown in Figure 4c and 4e. On the other hand, the two-phase approach gives higher priority to the profitability of loop permutation orders than minimizing dependence distances; loop fusion is applied to the interchanged loops with most profitable and legal permutation orders. For the case of 2mm type-1, the two-phase approach does not fuse any level of the two loop nests because of the conflict of the permutation order (Figure 4d). For 2mm type-2, the outermost loops after permutation have the same array reference pattern to the shared array `A` and are fused (Figure 4f).

	2mm type-1	2mm type-2
Pluto	95.39[sec]	41.50[sec]
Two-phase	31.33[sec]	34.13[sec]

TABLE I: Sequential execution time after loop tiling on Intel Xeon

Table I shows the sequential execution time on Xeon for type-1 and type-2 via Pluto and the two-phase approach. The Pluto version automatically applied parametric loop tiling [2] and we manually searched the best tile size. For the two-phase approach, parametric tiling as a part of the phase-2 transformation and the best tile search were manually processed. For both of 2mm-type1 and 2mm-type2, the proposed two-phase approach shows smaller execution time than Pluto because of the benefit to keep the most profitable loop order. In the future work, we will evaluate other benchmarks including parallel versions and platforms.

B. 2d-jacobi: 2-dimensional Stencil Computation Kernel

Figure 5 shows the input and transformed codes for 2d-jacobi. Figure 5b shows a pseudo code for Pluto, where loop peeling and tiling transformations are omitted so as to increase readability. Figure 5c shows the code after the phase-1 transformation and index set shifting to help loop skewing in the phase-2 transformation. Note that the above index set shifting can be included in the phase-1 algorithms in Section V-A. The difference between Figure 5b and Figure 5c is the application of loop skewing. Most of AST-based skewing algorithms rely on loop dependence vectors to select legal and profitable skewing factors. In the case of 2d-jacobi, if a skewing algorithm is given the dependence vectors $(0,1,1)$, $(1,-1,-1)$, $(0,1,0)$, $(1,-1,0)$, $(0,1,2)$, $(1,-1,-2)$, $(0,2,1)$, $(1,-2,-1)$, $(0,0,1)$ and $(1,0,-1)$, it can obtain the skewing factors `i1 = t`, `i2 = 2*t + i` and `i3 = 2*t + j`, which are equivalent to the Pluto transformation.

VII. Related Work

There is an extensive body of literature on the polyhedral model and AST-based transformations. In this section, we focus on past contributions that are most closely related to this paper.

Pluto [8] is a polyhedral framework for locality and parallelism optimizations, which handles the whole loop transformations by solving ILP formulation based on dependence distances. As combination with Pluto, an empirical search is employed in order to find better fusion/code motion transformations [17]. There is also an iterative search approach based on a cost model for vectorizations; it tests all legal transformations of loop permutation [22]. In the field of High Level Synthesis for FPGA implementation, there is an approach that identifies a shape of the preferred access function in the generated code and iteratively tests various transformations [24]. Several approaches also extend the polyhedral model to cost models, e.g., a cost model for loop fusion based on the prefetch streams (especially IBM Power chips) [7], and an estimator of the traffic between cache and main memory to drive better transformations [5].

AST-based transformation frameworks have a long history [15], [23]. There are a lot of pioneering works for parallelizing and locality optimizing compilers in both research and industry [10], [3], [19], [4], [21].

VIII. Conclusions

In this paper, we presented our early experiences with integrating polyhedral and AST-based transformations. As an initial demonstration of the benefit of the integration, we introduced a practical two-phase approach. For phase-1, we extended the polyhedral framework with a cost-based approach for locality optimizations. This framework focuses on loop fusion and permutation for inter-loop and intra-loop data locality optimizations, respectively. As the result of the phase-1 transformation, loops with data locality are legally fused/interchanged, and each fused loop nest is enabled for phase-2 that can support any AST-based transformation on a single loop nest. Our preliminary experiments showed how the proposed two-phase approach is applied to the benchmark programs, and demonstrated the benefits of the integrating approach relative to Pluto polyhedral framework for locality and parallelism optimizations.

For future work, we extend the fusion profitability analysis to support/integrate other analytical models than the DL cost model, and also to enhance the dependence vector analysis derived from the polyhedral dependence. The current implementation is a preliminary version; the full implementations of the whole transformation framework are also addressed in future work.

Acknowledgments

We are grateful to Louis-Noël Pouchet at University of California, Los Angeles for his feedback on this work and contributions to the underlying compiler framework used in this paper. We would like to thank P. Sadayappan at The Ohio State University and J. Ramanujam at Louisiana State University for early discussions on the polyhedral model and reuse distance analysis.

This work is an outgrowth of the PACE project supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915. We are especially grateful to the PACE team members at Ohio State University and Rice University who contributed to the implementation of the PACE compiler, since that implementation provided a starting point for this work.

References

[1] Polybench/c 3.2. *http://sourceforge.net/projects/polybench/*.

[2] Polyopt/c. *http://www.cs.ucla.edu/ pouchet/software/polyopt/polyopt.html*.

[3] The rice scalar compiler group. *http://www.cs.rice.edu/ keith/MSCP/*.

[4] Sun studio c/c++/fortran compiler. *http://developers.sun.com/prodtech/cc/compilers_index.html*.

[5] C. Bastoul and P. Feautrier. More legal transformations for locality. In *In Euro-Par10, number 3149 in LNCS*, pages 272–283. Springer Verlag, 2004.

[6] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303, Paphos, Cyprus, Mar. 2010. Springer Verlag.

[7] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proc. ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08)*, 2008.

[9] T. P. compiler project. http://pace.rice.edu.

[10] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on parallel and distributed systems*, 9(1), Jan. 1998.

[11] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[12] P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *IJPP*, 21(6):389–420, 1992.

[13] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness. *Proc. LCPC 91*, 589:328–343, 1991.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *IJPP*, 34(3):261–317, June 2006.

[15] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. 2001.

[16] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI13)*, Seattle, WA, June 2013. ACM Press.

[17] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010. IEEE Computer Society Press.

[18] ROSE compiler infrastructure. http://rosecompiler.org.

[19] V. Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. & Dev.*, 41(3), May 1997.

[20] J. Shirako, K. Sharma, N. Fauzia, L. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In *Proceedings of the 2012 International Conference on Compiler Construction (CC 2012)*, 2012.

[21] X. Tian et al. Intel OpenMP c++/fortran compiler for hyper-threading technology. *Intel Technology Journal*, 6, 2002.

[22] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *in PACT 09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337.

[23] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[24] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '13, pages 9–18, New York, NY, USA, 2013. ACM.

1. Loop Fusion:

```
for i = lw1, up1
  S1;
for i = lw2, up2
  S2;
```

⟹

```
for i = min(lw1,lw2), max(up1,up2)
  if (lw1 <= i && i <= up1) S1;
  if (lw2 <= i && i <= up2) S2;
```

2. Loop Distribution:

```
for i = lw, up
  S1;
  S2;
```

⟹

```
for i = lw, up
  S1;
for i = lw, up
  S2;
```

3. Loop Permutation/Interchange:

```
for i = lwi, upi
  for j = lwj, upj
    S;
```

⟹

```
for j = lwj, upj
  for i = lwi, upi
    S;
```

4. Loop Skewing (with skewing factor i2 = i & j2 = i+j):

```
for i = lwi, upi
  for j = lwj, upj
    S;
```

⟹

```
for i2 = lwi, upi
  for j2 = i2+lwj, i2+upj
    i = i2;
    j = j2 - i;
    S;
```

5. Index Set Shifting:

```
for i = lw, up
  S;
```

⟹

```
for i2 = lw+c, up+c
  i = i2 - c;
  S;
```

6. Loop Reversal:

```
for i = lw, up
  S;
```

⟹

```
for i = up, lw, -1
  S;
```

7. Loop Peeling (handling first iteration as prologue loop):

```
for i = lw, up
  if (i == lw) S1;
  else S2;
```

⟹

```
for i = lw, lw
  S1;
for i = lw+1, up
  S2;
```

8. Loop Strip Mining:

```
for i = lw, up
  S;
```

⟹

```
for is = lw, up, sz
  for i = is, min(is+sz-1,up)
    S;
```

9. Loop Tiling:

```
for i = lwi, upi
  for j = lwj, upj
    S;
```

⟹

```
for it = lwi, upi, Ti
  for jt = lwj, upj, Tj
    for i = it, min(it+Ti-1,upi)
      for j = jt, min(jt+Tj-1,upj)
        S;
```

10. Loop Unrolling (assuming `up-lw+1` is evenly divided by `sz`):

```
for i = lw, up
  S;
```

⟹

```
for iu = lw, up, sz
  i = iu + 0;
  S;
  ...
  i = iu + sz - 1;
  S;
```

11. Unroll-and-Jam (same assumption as unrolling):

```
for i = lwi, upi
  for j = lwj, upj
    S;
```

⟹

```
for iu = lwi, upi, sz
  for j = lwj, upj
    i = iu + 0;
    S;
    ...
    i = iu + sz - 1;
    S;
```

Fig. 6: Individual loop transformations