Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations

Jun Shirako
Rice University

Louis-Noël Pouchet
University of California Los Angeles

Vivek Sarkar
Rice University

***Abstract*—Optimizing compilers targeting modern multi-core machines require complex program restructuring to expose the best combinations of coarse- and fine-grain parallelism and data locality. The polyhedral compilation model has provided significant advancements in the seamless handling of compositions of loop transformations, thereby exposing multiple levels of parallelism and improving data reuse. However, it usually implements abstract optimization objectives, for example "maximize data reuse", which often does not deliver best performance, e.g., the complex loop structures generated can be detrimental to short-vector SIMD performance. In addition, several key transformations such as pipeline-parallelism and unroll-and-jam are difficult to express in the polyhedral framework. In this paper, we propose a novel optimization flow that combines polyhedral and syntactic/AST-based transformations. It generates high-performance code that contains regular loops which can be effectively vectorized, while still implementing sufficient parallelism and data reuse. It combines several transformation stages using both polyhedral and AST-based transformations, delivering performance improvements of up to $3\times$ over the PoCC polyhedral compiler on Intel Nehalem and IBM Power7 multi-core processors.**

I. Introduction

Optimizing compilers attempt to restructure the input program to extract the proper grain of parallelism and data locality that best fit the target architecture. For modern multi-core processors this amounts to: (1) exploiting data locality by grouping operations accessing the same or nearby data elements, to achieve both spatial and temporal locality; (2) exposing enough coarse-grain parallelism to effectively utilize all the available cores; and (3) exposing enough well-structured fine-grain parallelism to exploit the available short-vector SIMD units. Optimizing compilers are in charge of transforming the input program, typically using loop transformations, so that the generated code satisfies all three objectives. Such transformations include loop tiling and loop parallelization, but numerous complementary loop transformations may be needed first to make the tiling or parallelization possible.

The complexity of the loop transformation sequence needed to achieve both data locality and parallelism poses a significant challenge in the design of the optimization algorithm: previous work typically focused on one particular objective (parallelization, vectorization or data locality [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]) and attempts to find a sequence of loop transformations maximizing this objective, possibly by being detrimental to the other two objectives. To address this limitation, the polyhedral compilation framework restricts the class of programs it can manipulate to sequences of imperfectly nested loops with particular constraints on the loop bound and array subscript expressions. An immediate benefit is the ability to design optimization algorithms that can determine an arbitrarily complex sequence of loop transformations needed to enable tiling, coarse-grain parallelization and vectorization. In this unified framework, multiple objectives can be combined and prioritized in a single polyhedral formulation, ensuring the loop transformation sequence found implements the proper amount of tiling and parallelization. The benefit of this unified formulation has been best exemplified with the Pluto algorithm [14], [15], which has been successfully extended and specialized to integrate SIMD constraints [16].

Years of efforts to integrate the various optimization objectives in a single formulation have led to excessively complicated polyhedral loop transformations to be generated by such algorithms. In this work we make a compelling case for *decoupling the optimization problem into multiple stages*. Furthermore, we show that *several transformations should be performed outside of the polyhedral framework*, for both simplicity and performance. This apparently counter-intuitive approach is justified by key observations on the simpler code generated when using syntactic (e.g., AST-based) transformations instead of polyhedral transformations for certain stages such as exploiting pipeline-parallelism. In addition, cost modeling is simplified when dealing with different stages for different cost considerations. This in turn results in weaker constraints on the polyhedral transformation objectives, leading to simpler loop structures generated. As the final performance on modern multi-core chips is driven in large part by the effectiveness of SIMD vectorization, such simpler loops are more amenable to automatic vectorization by production compilers such as Intel ICC, leading to significant performance improvements of up to $3\times$ in our experiments over a fully-integrated state-of-the-art polyhedral compilation approach, which itself already widely outperforms native compilers such as Intel ICC and IBM XL/C.

This paper makes the following contributions. (1) We present a framework to perform end-to-end program optimization, with an integration of polyhedral and syntactic code transformations. (2) We extend current optimization heuristics used in polyhedral optimizers by targetting more sophisticated optimization goals than generating doall parallelism with minimum reuse distance. (3) We present a multi-stage optimization algorithm, tackling each individual performance goal (intra-tile data reuse, vectorizability, coarse-grain parallelism, inter-tile data reuse) with the proper set of loop transformations (polyhedral, syntactic, or both). (4) We report extensive performance results, showing significant performance improvement

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.29

```
1  for (i = 0; i < NI; i++) {
2    for (j = 0; j < NJ; j++) {
3  R:  tmp[i][j] = 0;
4      for (k = 0; k < NK; k++)
5  S:    tmp[i][j] += alpha * A[i][k] * B[k][j];
6    }
7  }
8  for (i = 0; i < NI; i++) {
9    for (j = 0; j < NL; j++) {
10 T:  D[i][j] *= beta;
11     for (k = 0; k < NJ; k++)
12 U:    D[i][j] += tmp[i][k] * C[k][j];
13   }
14 }
```

Fig. 1: Input 2mm code

```
1  for (c1 = 0; c1 < NI; c1++) {
2    for (c2 = 0; c2 < NI; c2++) {
3  R:  D[c1][c2] *= beta;
4  T:  tmp[c1][c2] = 0;
5      for (c7 = 0; c7 < NI; c7++)
6  S:    tmp[c1][c2] += alpha * A[c1][c7] * B[c7][c2];
7      for (c7 = 0; c7 <= c2; c7++)
8  U:    D[c1][c7] += tmp[c1][c2-c7] * C[c2-c7][c7];
9    }
10   for (c2 = NI; c2 <= 2 * NI -2 ; c2++)
11     for (c7 = c2 - NI + 1; c7 < NI; c7++)
12 U:    D[c1][c7] += tmp[c1][c2-c7] * C[c2-c7][c7];
13 }
```

Fig. 2: 2mm using maximal polyhedral fusion [15]

```
1  for (c1 = 0; c1 < NI; c1++) {
2    for (c3 = 0; c3 < NJ; c3++)
3  R:  tmp[c1][c3] = 0;
4    for (c3 = 0; c3 < NK; c3++)
5      for (c5 = 0; c5 < NJ; c5++)
6  S:    tmp[c1][c5] += alpha * A[c1][c3] * B[c3][c5];
7    for (c3 = 0; c3 < NL; c3++)
8  T:  D[c1][c3] *= beta;
9    for (c3 = 0; c3 < NJ; c3++)
10     for (c5 = 0; c5 < NL; c5++)
11 U:    D[c1][c5] += tmp[c1][c3] * C[c3][c5];
12 }
```

Fig. 3: 2mm using our flow

over classical polyhedral optimizers, by ensuring the final code properly exploits good vectorization, data locality through tiling, and doall/pipeline/reduction parallelism wherever possible.

The rest of the paper is organized as follows. Sec. II motivates our approach and provides an overview of our optimization strategy. Sec. III recalls key concepts of polyhedral compilation, and introduces our framework to select affine transformations. Sec. IV presents various AST-based optimizations and shows how to form an end-to-end optimizer mixing polyhedral and AST transformations. Extensive experimental results are presented in Sec. V before discussing the related work in Sec. VI and concluding in Sec. VII.

II. Motivation and Overview

To motivate our new optimization flow, we use 2mm as a driver example. 2mm is a benchmark from PolyBench/C [17] which computes a sequence of two matrix multiplications $C = A.B; E = D.C$. The original code is shown in Fig. 1. We applied a state-of-the-art polyhedral compiler that enables maximal data locality, tilability, and coarse-grain parallelization in a single optimization stage [15]. The resulting code structure (tiling is omitted for readability) is shown in Fig. 2. It corresponds to the output of maximal fusion when using the native Pluto algorithm [15] as implemented in the PoCC compiler [18]. In contrast, the loop structure (also with tiling omitted) generated by our proposed framework is shown in Fig. 3. Their performance on an Intel Nehalem and an IBM Power7 processor is shown in Table I.

Variants	Intel Nehalem	IBM Power7
original	2.4 GF/s	0.5 GF/s
PoCC	14 GF/s	29 GF/s
our flow	19 GF/s	62 GF/s

TABLE I: 2mm performance comparison

Maximal data locality as implemented in Fig. 2 translates to minimizing the temporal distance between two accesses to the same memory location. This is a core objective of the polyhedral optimizer to achieve good data reuse. This objective is expressed in the polyhedral framework using a minimization objective. We first formulate the expression of temporal reuse distance between two operations $\vec{x}$ and $\vec{y}$ which access the same memory location as:

$$|\Theta(\vec{x}) - \Theta(\vec{y})| \leq dist$$

where Θ is the function modeling the scheduling of operations, assigning a virtual timestamp to each operation $\vec{x}$ to be executed during the computation. By finding a schedule Θ which minimizes *dist* subject to data dependence constraints (which may require for instance that $\Theta(\vec{x}) > \Theta(\vec{y})$), we end up with a schedule of the operations that implements maximal data locality. This schedule is then implemented through a sequence of loops transformations on the original code automatically within the polyhedral framework. *A fundamental observation is that this minimization objective is too strong: there is no point in minimizing the reuse distance from a performance point of view*. The real performance objective is to ensure that the implemented reuse distance is such that when reused, the data will not have been evicted from the cache. This implies the need to properly model cache replacement. Another key observation relates to SIMD vectorization: the code implementing maximal reuse uses a triangular loop and a complex access (`c2-c7` at lines 8 and 12 in Fig 2) along the vectorizable dimension. Consequently, the SIMD performance achievable will be very limited.

Our proposed optimization flow addresses these deficiencies. To cope with the intricate modeling of the exact data reuse requirements, we decouple the optimization problem into stages.

The first stage selects an affine transformation modeling a combination of loop fusion, distribution and permutation, by using the DL (Distinct Lines) model, which was designed to estimate the number of distinct cache lines, or TLB entries, accessed in a loop nest [5]. The selection of the transformations to apply is driven by the DL cost model and by the goal of maximizing the number of "clean" inner-loops that can be effectively vectorized (corresponds to Fig. 3). Once an affine transformation for the code is computed, that amounts to implementing combinations of loop fusion / distribution / permutation / code motion / peeling / shifting, *we go out of the polyhedral transformation framework and perform a series of transformations in a syntactic manner on the program AST*. Skewing may be implemented to ensure tilability — both skewing and tiling are implemented directly on the AST representation. Our experience shows that the generated code from performing skewing and tiling on the AST has simpler loops than when implementing skewing and tiling in the polyhedral framework. Parallelization is a separate stage in our optimization flow, that is taken in isolation from the computation of the affine transformation. The goal of this separation is to tackle another key deficiency of integrated polyhedral optimization: that *as only doall parallelism is explicitly*

modeled in the polyhedral framework, pipeline parallelism is typically implemented as inefficient wavefront schedules. In our framework, instead of implementing a wavefronting of the tile loops which generates load imbalance because of the start-up / draining phases, we rely on a special pipeline-parallel construct that has been proposed as an OpenMP extension [19] which achieves the same parallelism but without the negative effects of wavefronting. An overview of the various stages is shown in Algorithm 1, where the optimization flow is broken into stages, each of which addresses a particular performance goal.

Algorithm 1: End-to-end algorithm

Input : Source program $P = (Poly, AST)$
begin
  P := fusion_and_permutation_with_DL($P.Poly$);
  P := skewing_for_tilability($P.AST$);
  P := coarse_grain_parallelization($P.AST$);
  P := tiling_for_locality($P.AST$);
  P := intra_tile_optimizations($P.AST$);
end
Output: Parallelized and optimized program P

To summarize, our flow has the following key properties. First, it relaxes the constraints on the polyhedral loop transformation stage by looking for an affine transformation that achieves enough data locality and preserves some parallelism. This is in contrast to previous work such as the Pluto algorithm which embeds tilability and parallelization upfront. As a result, the loop nest generated by our approach can be significantly simpler. Second, it uses a collection of AST-based transformations to ensure all performance goals (data reuse through tiling, inter-tile parallelization, intra-tile vectorization) are addressed by the optimization. Using AST-based transformation enables (1) the use of non-affine transformations, such as the pipeline-parallel OpenMP construct; (2) in general, a simpler loop structure than that generated using only polyhedral transformations, resulting in a more efficient auto-vectorization of the generated codes. We present in the following the details of the optimization flow in Secs III and IV, before presenting extensive experimental results.

III. Cache-Aware Affine Transformations

A. Background on the Polyhedral Model

The polyhedral model [20] is an algebraic framework for affine program representations and transformations. We rely on this framework for the first stage of our optimization flow, where the objective is to implement sufficient data reuse opportunities (in contrast to maximizing data locality) while preserving SIMD parallelism. The major strengths of the polyhedral model over AST-based approaches include: 1) the unified representation of perfectly and imperfectly nested loops with parametric bounds and a large set of transformations that can be performed on them [21], [22], and 2) mathematical approaches to express data dependences and to encode the legality of transformations [23], [21], [24]. These features provide a high degree of flexibility in specifying loop transformations in a unified framework.

The polyhedral model is a flexible and expressive representation for imperfectly nested loops with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoPs) [21], [22], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop iterators and variables that are invariant during the SCoP execution. Numerous scientific kernels exhibit those properties; they can be found in image processing filters, linear algebra computations, etc. [22]. We now describe the key data structures to represent programs.

Iteration domains: For each textual statement in the program the set of its run-time instances is captured with an integer set bounded by affine inequalities, intersected with an affine integer lattice [25], that is the iteration domain of the statement. Each point in this set represents a unique dynamic instance of the statement, such that the coordinates of the point corresponds to the value the surrounding loop iterators take when this instance is executed. For instance for statement `S` in Fig. 1, its iteration domain $\mathcal{D}_S$ is:

$$\mathcal{D}_S = \{(i,j,k) \in \mathbb{Z}^3 \mid 0 \leq i < \mathrm{NI} \wedge 0 \leq j < \mathrm{NJ} \wedge 0 \leq k < NK\}$$

We denote $\vec{x}_S \in \mathcal{D}_S$ as a point in the iteration domain.

Access functions: They represent the location of the data accessed by the statement. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of a read reference `A[i][k]` surrounded by 3 loops i, j and k is simply $f_A(i,j,k) = (i,k)$.

Data dependences: The sets of statement instances between which there is a data dependence relationship are modeled as equalities and inequalities describing a *dependence polyhedron*. This relationship is defined at the granularity of the array cell. If two instances $\vec{x}_R$ and $\vec{x}_S$ access the same array cell and at least one of these accesses is a write, then they are said to be in dependence. Therefore to respect the program semantics, the transformed program must ensure $\vec{x}_R$ and $\vec{x}_S$ are executed in the same order as in the original program. Given two statements R and S and a data dependence $R \rightarrow S$, a dependence polyhedron, written $\mathcal{D}_{R,S}$, contains all pairs of dependent instances $\langle \vec{x}_R, \vec{x}_S \rangle$. Given a set of statements, a polyhedral dependence (multi-)graph, PoDG, is defined as $G = (V,E)$ where V contains all statements and E contains one edge $R \rightarrow S$ per dependence polyhedra $\mathcal{D}_{R,S}$, $R,S \in V$, labeled by the dependence polyhedra.

Multiple dependence polyhedra may be required to capture all dependent instances, at least one for each pair of array references accessing the same array cell (scalars being a particular case of array). It is possible to have several dependence polyhedra per pair of textual statements, as some may contain multiple array references. In our work, all dependence polyhedra are automatically extracted from the program polyhedral representation, using the Candl tool [18].

Affine program transformations: An affine transformation captures, in a single step, what may typically correspond to a sequence of tens of textbook loop transformations [22]. It takes the form of a carefully crafted affine multidimensional schedule represented as a matrix. A schedule is a function which associates a logical execution date (a timestamp) to

each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the target program, statement instances will be executed according to the increasing lexicographic order of their timestamp. To construct a full program optimization, we build a collection of schedules $\Theta = \{\Theta^{S1}, \ldots, \Theta^{Sn}\}$, that is a list of the statement scheduling function for each statement in the program, such that for all dependent instances the producer instance is scheduled before the consumer one. In this work, we focus on the affine program transformation stage for finding a schedule that implements good data locality while preserving SIMD parallelism opportunities, as explained in Sec. III-C. By focusing the objective carefully, we can effectively restrict the set of transformations we consider to a subset of acceptable polyhedral transformations. More precisely, we look for a composition of:

- Multidimensional fusion / distribution / code motion of the statements, to implement good data locality and preserve parallelism opportunities.
- Multidimensional retiming (a.k.a. index set shifting) and loop permutation (a.k.a. interchange), to realign memory accesses for better locality, and/or make the fusion/distribution chosen legal.

Given a statement S, we model compositions of the transformations above with an affine form of the d enclosing loop iterators i_1,...,i_d such that (1) the matrix describes a one-to-one function using exclusively integer coefficients; and (2) the number of rows in the matrix is $2d+1$, where each odd row models a constant function [22]. Such a schedule can then directly be given to a polyhedral code generator such as CLooG [25] to implement the desired transformation. Such a schedule can be expressed as follow:

$$\Theta^S(\vec{x}_S) = \begin{pmatrix} 0 & \ldots & 0 & \beta_1 \\ \alpha_{1,1} & \ldots & \alpha_{1,d} & c_1 \\ 0 & \ldots & 0 & \beta_2 \\ \alpha_{2,1} & \ldots & \alpha_{2,d} & c_2 \\ \vdots & & & \vdots \\ 0 & \ldots & 0 & \beta_d \\ \alpha_{d,1} & \ldots & \alpha_{d,d} & c_d \\ 0 & \ldots & 0 & \beta_{d+1} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_d \\ 1 \end{pmatrix}$$

where $\forall k \in \{1..d\}$, $\sum_{i=1}^{d} |\alpha_{k,i}| = 1$ and $\sum_{i=1}^{d} |\alpha_{i,k}| = 1$. This encoding is reminiscent of classical 2d+1 encoding [26], [16] where β coefficients are used to model multidimensional statement interleaving (i.e., multidimensional fusion/distribution/-code motion), the α coefficients with the property above model loop permutation and loop reversal, and the c coefficients model multidimensional retiming (i.e., index set shifting).

For example, the code in Fig. 3 is obtained from Fig. 1 by using four schedules, one for each statement, and the schedule for statement `T` is:

$$\Theta^T(\vec{x}_T) = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 2 \\ j \\ 0 \end{pmatrix}$$

The form of affine schedule described above is a restriction on more general affine scheduling [24]. However, the key benefits of this representation outweigh its limitations. First and foremost, *such schedules are invertible*, which implies that Θ^{-1} is well defined and can be used in the optimization algorithm to reason about the array access functions after transformations, without having to generate the code implementing the transformation. We also remark that by construction each even row is linearly independent to the other even rows. Second, each even row - i.e., loop dimension - corresponds to one of original loop iterators; the transformed loops naturally keep the original access pattern (or its reversal) and are more amenable to SIMD vectorization.

B. DL Model for profitability analysis

The DL (Distinct Lines) model was designed to estimate the number of distinct cache lines, or TLB entries, accessed in a loop nest [27], [5]. In this section, we briefly introduce how this analytical model is used as the memory cost model that guides loop fusion and permutation in the proposed framework. In contrast to reuse distance minimization algorithms like Pluto [28], this cost model captures spatial data locality in addition to temporal locality.

Based on machine parameters, e.g., cache line size and TLB page size, and program parameters, e.g., array dimension size and access function, the DL model expresses the number of distinct lines on a given cache/TLB as a function of enclosing loop sizes [27], [5]. In the following discussion, we assume loop tiling to be applied and DL is a function of tile sizes - i.e., $DL(t_1, t_2, \cdots, t_d)$. Figure 4 shows a simple case with two array references enclosed in a triply nested tiled loops.

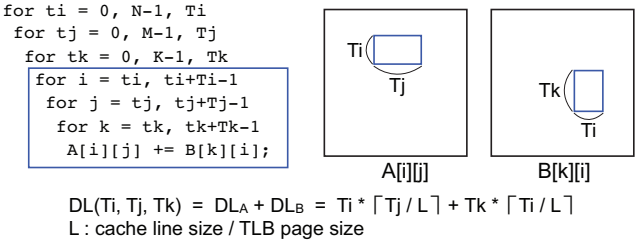

Fig. 4: Example of DL for tiled loop nest

The per-iteration memory cost of a given loop nest on a specific cache/TLB is defined as follow.

$$mem\_cost(t_1, t_2, \cdots, t_d) = \frac{Cost_{line} \times DL(t_1, \cdots, t_d)}{t_1 \times t_2 \times \cdots \times t_d}$$

$Cost_{line}$ represents the memory cost (miss penalty) per line on cache/TLB of interest. Under the assumption that the cache/TLB keeps any data until the last reuse, $Cost_{line} \times DL$ represents the total cost to bring all data of the loop nest on the cache/TLB. The following profitability analyses suppose the proper application of loop tiling in the latter AST-based phase and data per tile is confined to cache/TLB capacity. Although the applicability of tiling depends on loop dependences and other transformations, optimistic assumptions are generally acceptable for guiding profitability analyses. In the following, *DL* and *mem_cost* are used as functions of tile sizes.

1) Best Permutation Order Analysis: The partial derivative of *mem_cost* with respect to tile size t_i, $\delta mem\_cost/\delta t_i$, represents the variation rate of memory cost when increasing t_i. $\delta mem\_cost/\delta t_i < 0$ indicates that increasing t_i causes a decrease in memory cost and placing the loop with the most negative value of $\delta mem\_cost/\delta t_i$ at the innermost position could yield the largest benefit on data locality. In our approach, $\delta mem\_cost/\delta t_i$ is used as the priority for loop permutation - i.e., the ascending order of $\delta mem\_cost/\delta t_i$ is the most profitable permutation order (from inner to outer).

2) Loop Fusion Profitability Analysis: The DL memory cost is also used as a metric of loop fusion profitability by comparing *mem_cost* before and after fusion. In general, proper loop fusion enhances inter-statement data reuse while reduces upper bounds of tile sizes that fit within cache/TLB capacity. These boundaries can be analytically estimated by an extension of DL model [29] and the minimum *mem_cost* within the boundaries is computed to check whether fusion can reduce the minimum *mem_cost*. Note that the profitability of fusion will be affected by multiple factors including data reuse via cache/TLB/register, parallelism, and prefetching. The current criteria for applying loop fusion are listed in Algorithm 5 in Sec. III-C.

C. Optimization Algorithms

This section describes the proposed algorithms to compute the various coefficients of the scheduling matrices. The overall algorithm is shown in Algorithm 2, which starts from the top loop level ($k = 1$ with S as all statements in the target program scope) and recurses until all schedules for S become one-to-one mapping. Note that the number of rows of the produced schedules may be more than 2d+1 but such schedules are always convertible to a 2d+1 form. A polyhedral dependence graph, PoDG, captures all data dependences among statements (Sec. III-A). All edges in the PoDG - i.e., dependence polyhedra - are tagged as unsatisfied before starting Algorithm 2.

Algorithm 2: Affine transformation

Input : S : set of statements S_i,
PoDG : polyhedral dependence graph,
k : current nest level, or dimension,
$niter^{S_i}$: # iterators not yet scheduled in Θ^{S_i}

```
1  begin
2      PoDG' := subset of PoDG w/o satisfied dependence;
3      SccSet := compute SCCs of PoDG';
4      // Intra-SCC transformation (permutation)
5      for each Scc_a ∈ SccSet do
6          if ∃S_i ∈ Scc_a : niter^{S_i} ≥ 1 then
7              compute permutation and get constraints on
               reversal (α_{k,*}) and retiming (c_k) at level k
               for statements in Scc_a (Algorithm 4);
8      // Inter-SCC transformation (fusion)
9      FuseSet := compute β_k and get constraints on
       reversal and retiming for statements in S
       (Algorithm 5);
10     for each Fuse_a ∈ FuseSet do
11         solve constraints on reversal and retiming from
           Algorithms 4, 5 and compute α_{k,*} and c_k for
           statements in Fuse_a;
12         // Recursive process for next dimension
13         if ∃S_i ∈ Fuse_a : niter^{S_i} ≥ 1 then
14             done := false;
15             if Fuse_a is a SCC at level k then
16                 done := Algorithm3(Fuse_a, PoDG, k+1);
17             if !done then
18                 Algorithm2(Fuse_a, PoDG, k+1);
19         else if ∃S_i ∈ Fuse_a : niter^{S_i} = 0 then
20             compute β_{k+1} of stmt. in Fuse_a (Algo. 5);
21 end
```

Output: Dimensions $k..m$ of schedules Θ^{S_i} for all statements

Algorithm 2 first computes the Strongly Connected Components (SCC) of statements in S at loop level k based on unsatisfied dependence edges - i.e., dependences not carried by the outer dimensions - in *PoDG* and stores them into *SccSet* (lines 2–3). For each Scc_a in *SccSet*, Algorithm 4 is applied and computes a permutation for level k, so that all statements in Scc_a are fused at level k (lines 5–7). Note that this process always succeeds because at least the original loop order satisfies all dependences among Scc_a. Algorithm 5 computes the fusion/distribution at level k, for all statements in S based on profitability and legality of fusion (line 9). The statements with identical β_k value are fused, and grouped into $Fuse_a \in FuseSet$. Algorithms 4 and 5 also provide the constraints on loop reversal (the sign of α) and multidimensional retiming (c_k) to ensure only legal transformations are considered. For each statements in $Fuse_a$ reversal and retiming are computed, under polyhedral legality constraints [21], [30], and to minimize the cost functions of reuse distance and skewing factor (line 11). After computing the schedule for statements in $Fuse_a$ at level k, it recursively processes the next dimension $k+1$ if $Fuse_a$ contains a statement such that all loop iterators are not yet scheduled in the schedule Θ^{S_i} until current level k (line 13). If $Fuse_a$ is a single SCC at level k (line 15), $Fuse_a$ is passed to Algorithm 3 that tries to perfectly fuse all statements so as to enable loop tiling and leverage data locality of the loop dimension at level k (line 16). This process can fail while Algorithm 2 always succeeds (line 18). Finally, β_{d+1}, the statement order at the innermost loop body, is computed by Algorithm 5 (lines 19–20).

Algorithm 3 computes the schedules at the given nest level k as with Algorithm 2. The difference is that all statements in S are passed to Algorithm 4 and legal $\alpha_{k,*}$ to fuse all statements are searched for (line 2). This process does not always succeed because S can contain multiple SCCs. The solutions of deeper nest levels are recursively explored (line 9); Algorithm 3 succeeds only when all solutions until the innermost dimension are successfully found.

Algorithm 4 computes $|\alpha_{k,*}|$, the loop permutation at level k, for the given statements S so that Θ_{2k}, the $2k$-th row of the schedule, is legal and as close to the best permutation order by the DL model (Sec. III-B1) as possible. Let d denote the number of loop iterators enclosing S_i and P^{S_i} denote the $d \times d$ permutation matrix corresponding to the most profitable permutation order by DL model for statement S_i, e.g., the loop iterators to be located at the outermost and innermost

Algorithm 3: Affine transformation for perfect fusion

Input : S : set of statements S_i,
$PoDG$: polyhedral dependence graph,
k : current nest level, or dimension,
$niter^{S_i}$: # iterators not yet scheduled in Θ^{S_i}

1 **begin**
2 $succeed$:= compute permutation and get constraints reversal and retiming for statements in S (Algo. 4);
3 **if** !$succeed$ **then**
4 return $false$;
5 **for** each $S_i \in S$ **do**
6 $\beta_k^{S_i} := 0$; // All statements are fused at level k
7 solve constraints from Algorithms 4 and compute reversal and retiming for statements in S;
8 **if** $\exists S_i \in S : niter^{S_i} \geq 1$ **then**
9 $done$:= Algorithm3(S, $PoDG$, k+1);
10 **if** !$done$ **then**
11 return $false$;
12 **else if** $\exists S_i \in S : niter^{S_i} = 0$ **then**
13 compute β_{k+1} of statements in S (Algorithm 5);
14 **end**

Output: Dimensions $k..m$ of the scheduling matrices, or return $false$ if no solution is found

Algorithm 4: Affine loop permutation

Input : S : set of statements S_i,
$PoDG$: subset PoDG w/o satisfied dependence,
k : current nest level, or dimension,
P^{S_i} : best permutation matrix of S_i by DL

1 **begin**
2 **repeat**
3 $hash$:= get next combo of row numbers for P;
4 **if** $hash = \emptyset$ **then**
5 return $false$;
6 **for** each $S_i \in S$ **do**
7 $r := hash(S_i)$; // current row number for P^{S_i}
8 **for** each $j \in \{1..d^{S_i}\}$ **do**
9 $|\alpha_{k,j}^{S_i}| := P_{r,j}^{S_i}$;
10 // Sign of $\alpha_{k,j}^{S_i}$ (i.e., reversal) is unfixed
11 $C := \emptyset$;
12 **for** each dependence in $PoDG$ **do**
13 cr := build legality constraints on reversal (sign of $\alpha_{k,*}$) and retiming (c_k);
14 $C := C \cap \{cr\}$;
15 **until** solutions of $\alpha_{k,*}$ and c_k for C exist ;
16 **end**

Output: Loop permutation at level k and constraints C on reversal and retiming, or return $false$ if no solution is found

are respectively given by inner products $P_1^{S_i} \cdot \vec{i^{S_i}}$ and $P_d^{S_i} \cdot \vec{i^{S_i}}$. Let $hash$ denote a hash table whose key is $S_i \in S$ and value is $r \in \{1, \cdots, d\}$ that points a row of P^{S_i}. At each round of the Algorithm 4, $hash$ is updated by a new combination of row numbers (line 3). Note that $hash$ has smaller values - i.e., points iterators to be located outer - at earlier round since the schedules are computed from the outermost dimension. If all combinations are tested, Algorithm 4 returns false to indicate there is no possible solution (lines 4–5). A combination of permutations is set to $|\alpha_{k,*}|$ for all statements based on the P and $hash$ (lines 6–10); the legality constraints of all unsatisfied dependence in the PoDG for current permutations are collected (lines 11–14). If there exist a valid retiming and/or reversal that satisfies all constraints C (line 15), then the current permutations $|\alpha_{k,*}|$ is legal and Algorithm 4 returns all computed information, $|\alpha_{k,*}|$ and constraints C on the sign of the non-null component of $\alpha_{k,*}$ (i.e., reversal) and on c_k (i.e., retiming).

Algorithm 5: Affine loop fusion/distribution

Input : S : set of statements S_i,
$SccSet$: set of SCCs Scc_a,
$PoDG$: subset PoDG w/o satisfied dependence,
k : current nest level, or dimension

1 **begin**
2 $C := \emptyset$;
3 $FuseSet := \emptyset$;
4 **repeat**
5 Scc_a := SCC of largest dimensionality in $SccSet$;
6 $SccSet := SccSet - \{Scc_a\}$;
7 $Fuse_a := Scc_a$; // set of statements to be fused
8 **repeat**
9 **for** each $Scc_b \in SccSet$ **do**
10 **if** precondition($Fuse_a$, Scc_b) **then**
11 $C_l := \emptyset$;
12 $PoDG'$:= subset of $PoDG$ whose edges connect $Fuse_a$ and Scc_b;
13 **for** each dependence in PoDG' **do**
14 cr := build legality constraints on reversal and retiming;
15 $C_l := C_l \cap \{cr\}$;
16 **if** solutions for C_l exist $\wedge$ parallelcondition($Fuse_a$, Scc_b) **then**
17 $SccSet := SccSet - \{Scc_b\}$;
18 $Fuse_a := Fuse_a \cup Scc_b$;
19 $C := C \cup C_l$;
20 **until** $Fuse_a$ is unchanged during the iteration ;
21 $FuseSet := FuseSet \cup \{Fuse_a\}$;
22 **until** $SccSet = \emptyset$;
23 compute β_k of statements in S based on $FuseSet$ and inter-SCC dependences;
24 **end**

Output: β_k, constraints C on reversal and retiming, and $FuseSet$ to keep sets of statements to be fused

After Algorithm 4 computes a permutation so that statements in a SCC may be fused at level k, Algorithm 5 computes which SCCs in $SccSet$ are fused/distributed at level k and implements the fusion in the schedule via the β_k coefficients for all statements in S. At each round of the Algorithm 5, Scc_a, the SCC that contains a statement with the largest dimensionality, is popped from $SccSet$ and used as the initial set of statements to be fused, $Fuse_a$ (lines 5–7). In the manner

of a greedy algorithm, it finds the maximum set of SCCs that can be legally and profitably fused to $Fuse_a$ (lines 8–20).

For $Scc_b \in SccSet$, we currently employ the following conditions to fuse Scc_b and $Fuse_a$. (1) Scc_b is a direct predecessor/successor of $Fuse_a$ or no dependences between $Fuse_a$ and Scc_b except for input dependence (legality precondition at line 10). (2) $\exists(S_i, S_j) \in Fuse_a \times Scc_b : S_i$ and S_j refer to an array via same access function for 1 to k dimensions, - i.e., sub-matrices of 1 to k columns are equivalent between access matrices $f^{S_i}\Theta^{S_i}{}^{-1}$ and $f^{S_j}\Theta^{S_j}{}^{-1}$ (profitability precondition at line 10). (3) As discussed in Sec. III-B2, fusion reduces DL memory cost (profitability precondition at line 10). (4) There exists a legal combination of reversal and retiming on unsatisfied dependences of the PoDG to fuse $Fuse_a$ and Scc_b. (lines 11–16). (5) The fusion does not kill outermost parallelism, i.e., neither of $Fuse_a$ nor Scc_b are outermost parallel loop $\vee$ the fused loop is also parallel (parallelcondition at line 16).

Condition (2) checks whether the reuse distance between two array accesses is constant. Condition (4) regarding parallelism is evaluated based on Δ_e, the dependence vectors of the transformed code, as discussed in Sec. IV-A. After the above greedy process, $Fuse_a$, the set of statements to be fused, is added to *FuseSet* (line 21). Finally, β_k for all statements in S is computed so as to satisfy all inter-SCC dependences and fusion grouping of *FuseSet* (line 23); all the computed information, β_k, constraints C on reversal and retiming, and grouping of *FuseSet* are returned.

IV. AST-based Transformations

After the above polyhedral transformation, typically the statements with data locality are fused into the same loop nest whose loops are legally and profitably permuted as needed. Such a loop nest is a good target for AST-based transformations. Loop dependence information is captured in the form of loop dependence vectors, which is the base of legality analysis of our AST-based loop transformation stage. These can be derived from the polyhedral framework as needed.

This section introduces an AST-based transformation framework that applies a sequence of individual loop restructurings to each loop nest. This AST-based framework can be categorized into three stages: parallelism detection, loop tiling, and intra-tile optimizations.

A. Detection of Parallelism

Loop level parallelism can be classified into three kinds: doall parallelism - i.e., no dependence among loop iterations, pipeline parallelism - i.e., all loop dependences are uniform and convertible into wavefront doall, and reduction parallelism - i.e., all loop carried dependences are due to commutative and associative computations. Note that reduction parallelization is widely supported both in research and academic compilers; we employed a standard approach to detect reduction parallelism based on pattern recognition of commutative and associative computations. This section briefly introduces a simple parallelism detector in the proposed framework.

Dependence vectors are a standard data dependence representation [1] which offers sufficient accuracy for our parallelism detector. While the previous affine-based transformation algorithm leveraged dependence polyhedra for modeling data dependences, offering the finest level of precision, limiting our attention to dependence vectors for our AST-based transformation framework is motivated by the lack of requirement for AST code to have affine control. *Since AST-based transformations do not require the loops to have affine control, they can be applied on a broader class of programs than polyhedral transformations*. Dependence vectors can be extracted seamlessly from dependence polyhedra if the code is affine, or be computed by other (sometimes conservative) analysis for non-affine codes [1].

Given a *loop* at level k in a loop nest, the detector first collects all dependences vectors Δ^e such that its source and target statements are included in *loop* and Δ^e is not satisfied by the outer loops - i.e., $\forall l < k : \Delta^e{}_l = 0$. Further, let Δ^{reduce} be the set of the dependence vectors derived from reduction computation in *loop*. According to the definition of each parallelism, the kind of parallelism for *loop* is determined in the following manner.

- *doall* if $\forall \Delta^e : \Delta^e{}_k = 0$
- *pipeline* if $\forall \Delta^e : \Delta^e{}_k = 0 \ \vee \ (\Delta^e{}_k \geq 1 \wedge \Delta^e{}_{k+1} \geq 0)$ [1]
- *reduction* if $\forall \Delta^e : \Delta^e{}_k = 0 \ \vee \ (\Delta^e{}_k \geq 1 \wedge \Delta^e \in \Delta^{reduce})$
- *reduction & pipeline* if $\forall \Delta^e : \Delta^e{}_k = 0 \ \vee \ (\Delta^e{}_k \geq 1 \wedge \Delta^e{}_{k+1} \geq 0) \ \vee \ (\Delta^e{}_k \geq 1 \wedge \Delta^e \in \Delta^{reduce})$

The second condition for pipeline parallelism intends that the loop nest has at least two-level pipeline parallelism, i.e., nest levels k and $k+1$. The fourth condition captures the cases where a loop has some dependence vectors to be handled as reduction and others to be handled as pipeline parallelism.

In our approach, the loop parallelism at the outermost possible level is always used regardless of kind - i.e., doall, pipeline or reduction - after data locality optimizations by the polyhedral phase. This strategy contrasts with other polyhedral frameworks which focus on implementing outermost doall parallelism. Figure 5 shows example output codes via those different approaches.

• Poly+AST approach

```
//doall parallel
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      A[i][j] = alpha * B[i][j];
//reduction parallel
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      S[j] += alpha * X[i][j];
//pipeline parallel
  for (i = 0; i < N; i++)
    for (j = 1; j < N-1; j++)
      C[i][j] = 0.33 * (C[i-1][j]
            + C[i][j] + C[i+1][j]);
```

• doall-only approach

```
//doall parallel
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      A[i][j] = alpha * B[i][j];
//doall parallel
  for (j = 0; j < N; j++)
    for (i = 0; i < N; i++)
      S[j] += alpha * X[i][j];
//doall parallel
  for (j = 1; j < N-1; j++)
    for (i = 0; i < N; i++)
      C[i][j] = 0.33 * (C[i-1][j]
            + C[i][j] + C[i+1][j]);
```

Fig. 5: Examples of parallelized outputs

The polyhedral phase in our approach selects the loop order of `i`, `j` for all three examples because of temporal and/or spatial data locality; the outermost doall, reduction and pipeline parallelisms are respectively used. On the other hand, other approaches considering only doall parallelism apply

[1] $\Delta^e{}_{k+1} \geq 0$ assumes the application of skewing prior to parallelization.

loop permutation for the second and third examples so as to implement outermost doall parallelism although such permutations could affect per-thread data locality via cache/TLB and vectorization efficiency[2].

B. Loop Tiling

Loop tiling is an important transformation to enhance temporal and spatial data reuse in memory hierarchies. In the proposed framework, the loop nest candidate for tiling can possibly contain dependence vectors with negative elements and hence not be fully permutable. To increase permutability (i.e., tilability[3]), our framework supports loop skewing as a preprocessing for loop tiling so as to make all dependence vector elements non-negative if possible. We employed a simple loop skewing algorithm to determine skewing factors starting from the outermost loop to innermost so that the loop nest does not contain loop dependence vectors with negative elements. As shown in Algorithm 1, skewing is applied prior to parallelization phase in Sec. IV-A.

Once the code has been skewed as needed to ensure that we have at least two loops with forward dependences only, the code is syntactically tiled along these now-permutable loops. We use a fixed tile size and limit to implementing a single-level rectangular tiling. While loop tiling is a transformation that can be easily implemented in the polyhedral model, our experience shows that performing a simple syntactic tiling of the loops by implementing a combination of AST-level strip-mining and interchange may deliver a slightly simpler loop structure than when using PoCC [18]. We observe that numerous syntactic aspects of the tiled code generated by PoCC can be refined thanks to the syntactic simplifications implemented in the polyhedral code generator CLooG [25], but we did not experiment with this tuning of CLooG.

C. Intra-tile Optimizations

While the loop structure (i.e., permutation order and fusion/distribution) of tiling loops must obey the polyhedral transformation in Sec. III-C, there is flexibility to employ additional transformations within individual tiles. For instance, the PoCC framework may permute intra-tile loops in the generated code using a basic inner-most loop SIMD profitability model.

In our framework, we currently support register tiling - i.e., multi-level loop unrolling - as an intra-tile optimization. Register tiling is an important optimization to enhance register reuse by increasing the number of statements in the loop body. Our experiments showed that up to 2× additional performance improvement can be obtained by register tiling. In the current implementation, the unrolling factor of each nest level is found by empirical search, though this approach can be improved by leveraging past work on selection of unroll factors. Loops within a tile are unrolled when they are permutable and have rectangular iteration space.

Other intra-tile optimizations such as additional permutation and distribution to enhance SIMD parallelism will be addressed in future work. Likewise, parametric loop tiling and explorations of optimal tile sizes and unrolling factors in our framework are subjects for future work.

[2]Vectorization can be enhanced by the additional per-tile optimization if loop tiling is applicable (see Sec. V).

[3]This paper focuses on rectangular tiling.

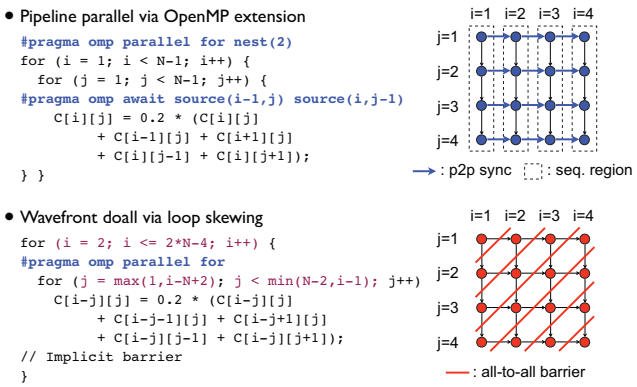

Fig. 6: Pipeline parallel vs. wavefront doall

D. Runtime Support for Reduction and Pipeline Parallelism

Based on the detected parallelism per loop, additional runtime support may be required. The OpenMP standard already supports scalar/array reductions for FORTRAN and scalar reductions for C language. We employ two extensions to OpenMP: support of array reductions in C [31] and support of cross-iteration synchronizations for pipeline parallelism [19]. The challenge for array reductions in C is the size detection of reduction arrays, which can be analyzable based on the array access function and domain information per statement - i.e., the upper/lower bound of each array dimension can be represented in a symbolic expression of global parameters. There is a proposal to support pipeline parallelism via point-to-point synchronizations in the OpenMP standard [19], which we used in our framework. It is well-known fact that pipeline parallelism has generally better scalability than wavefront doall due to synchronization efficiency and data locality, and this is confirmed by our experiments below.

V. Experimental Results

A. Experimental Protocol

Machines Experimental results presented in this paper have been obtained on two Linux-based SMP systems: a dual quad-core 2.8GHz Intel Core i7 (*Nehalem*) and a quad eight-core 3.86GHz IBM Power7 (*Power7*). On *Nehalem*, all experimental variants were compiled with Intel C/C++ compiler 12.0 with options of "-fast -xHOST" for sequential run, "-fast -xHOST -parallel" for Intel auto-parallelization, and "-fast -xHOST -openmp" for the Pluto and poly+ast (our work) auto-parallelizations. On *Power7*, all variants were compiled by IBM XL C/C++ compiler 11.1 with options of "xlc -O5" for sequential, "xlc_r -O5 -qsmp=auto" for IBM auto-parallel, and "xlc_r -O5 -qsmp=omp" for Pluto and poly+ast.

Benchmarks We used PolyBench/C 3.2 [17] for the performance evaluation. Table II shows the description of the representative set of PolyBench/C benchmarks we evaluated in our paper. We used the standard dataset for PolyBench both on *Nehalem* and *Power7* with the following exception. For fdtd-2d, jacobi-1d-imper, jacobi-2d-imper, and seidel-2d

(benchmarks with pipeline parallelism), we used the large dataset so as to provide sufficient parallelism.

Benchmark	Description
2mm	2 Matrix Multiplications (D=A.B; E=C.D)
3mm	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
fdtd-2d	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
jacobi-1d-imper	1-D Jacobi stencil computation
jacobi-2d-imper	2-D Jacobi stencil computation
mvt	Matrix Vector Product and Transpose
seidel-2d	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
trisolve	Triangular solver

TABLE II: Evaluated benchmarks in PolyBench

Experimental variants We implemented the polyhedral and AST-based transformation framework as a part of the PACE compiler framework [32]. We compared the proposed approach with the native compilers and pocc (the Polyhedral Compiler Collection) version 1.3, which is a flexible source-to-source iterative and model-driven compiler, embedding many of the state-of-the-art tools for polyhedral compilation [18].

We evaluated the following six experimental variants for each benchmark. (1) *icc-auto / xlc-auto* represents the automatic parallelization by Intel / XL compiler respectively. (2) *pocc* is the integrated polyhedral optimization supported in pocc, which also enables loop tiling for locality optimization. It implements the Pluto algorithm [15] using the smart-fuse heuristic for fusion, performs tiling and coarse-grain parallelization of tiles, possibly using a wavefront tile schedule if no outer-tile loop is parallel in the code. (3) *pocc vect* employs optimizations for efficient vectorization after the parallelization and localization by *pocc* including the application of additional loop permutation to the tiled code so as to place a good vector loop candidate at the innermost position in the tile. (4) *iterative* is the iterative compilation approach supported in pocc; this approach explores legal transformation space for outer-most fusion/distribution and also support loop tiling [26]. The number of generated variants per benchmark varies between 1 and 176, and auto-tuning has been performed on the target machine by running all variants. The *iterative* entry reports the performance achieved by the best variant found through auto-tuning. (5) *iterative vect* is the iterative approach with the complementary PoCC optimization for SIMD vectorization. (6) *poly+ast* is the polyhedral and AST-based transformation framework proposed in this paper, implementing the optimization algorithm shown in Alg. 1.

All experimental variants support loop tiling and we used the tile size of 32 for each tilable dimension for all benchmarks except for fdtd-2d, jacobi-1d-imper, jacobi-2d-imper, and seidel-2d, where 5 is used for the outer time-tile size. In addition for all variants we applied 2-level register tiling to the tiled loops as discussed in Section IV-C. When applicable, the innermost and second innermost loops within tiles are unrolled by factor of 1, 2, 4, 6 or 8; the performance numbers with the best unrolling factors are reported in the following sections[4].

B. Performance Results on Nehalem with 8 cores

The performance in GFLOP/s on *Nehalem* using 8 cores are shown in Figures 7, 8, 9; we divided all benchmarks into three figures based on the major source of parallelism[5].

Figure 7 shows the performance of the benchmarks whose major source of parallelism is doall. The experimental variants using the polyhedral model, *pocc*, *pocc vect*, *iterative*, *iterative vect*, and *poly+ast*, show better performance than *icc-auto* except for gesummv. For the variants of the pocc framework, the iterative compilation approaches - i.e., *iterative* and *iterative vect* - show equivalent or higher performance than model-driven compilation approaches - i.e., *pocc* and *pocc vect*; the *vect* optimization for efficient vectorizations also brings additional performance improvements except for syrk and syr2k.

On the other hand, the proposed model-driven compilation approach, *poly+ast*, shows higher performance than model-driven *pocc* and *pocc vect* and iterative compilation *iterative* and *iterative vect* for 2mm, 3mm, doitgen, and gemm and equivalent performance for other benchmarks. The major difference between *poly+ast* and pocc variants are inter-tile loop orders. Although the pocc variants can permute loops within a tile so as to improve the vectorization efficiency, permuting tiling loops is impossible at this stage. On the other hand, the polyhedral phase of *poly+ast* chooses the profitable loop order based on the DL memory cost and then loop tiling and intra-tile optimizations are applied later in the AST-based phase. As a consequence, the transformed codes by *poly+ast* are amenable to inter-tile data locality on multiple levels of cache and TLB. This leverages the advantages of both polyhedral and AST-based transformations in a synergistic manner.

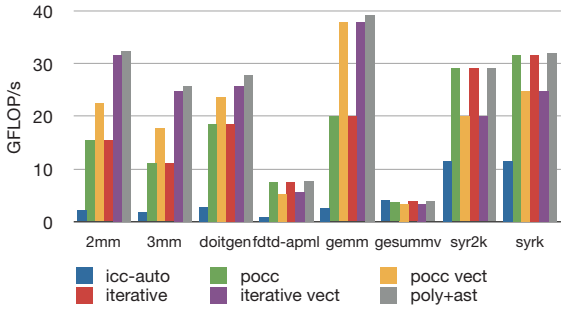

Fig. 7: PolyBench on 8-core Nehalem (doall parallelism is dominant)

[4]Due to the huge search space of iterative approach, the auto-tuning without register tiling was first performed and then register tiling was applied.

[5]There are some combinations of benchmarks and experimental variants which could not be compiled or executed.

	pipeline parallelism							reduction parallelism						
	adi	cholesky	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	trisolv	atax	bicg	correl	covar	gemver	mvt	symm
nehalem off	2.31	8.26	15.86	10.63	12.01	11.21	3.36	2.61	2.60	12.75	12.87	3.45	2.47	7.36
nehalem on	2.70	12.87	22.30	11.66	17.81	11.36	3.48	6.65	6.91	13.68	13.74	6.57	7.28	9.01
power7 off	3.43	29.45	25.50	7.89	15.31	15.92	3.26	1.74	1.74	22.66	22.38	4.24	1.44	25.94
power7 on	6.34	29.39	26.20	17.34	16.76	15.99	4.06	8.87	8.88	11.85	11.83	17.05	8.94	14.40

TABLE III: Performance when turning off/on runtime supports for pipeline and reduction (GFLOP/s)

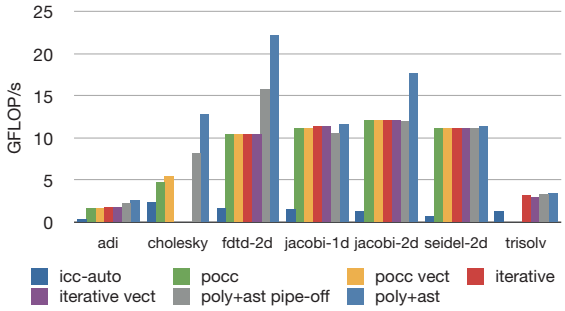

Fig. 8: PolyBench on 8-core Nehalem (pipeline parallelism is dominant)

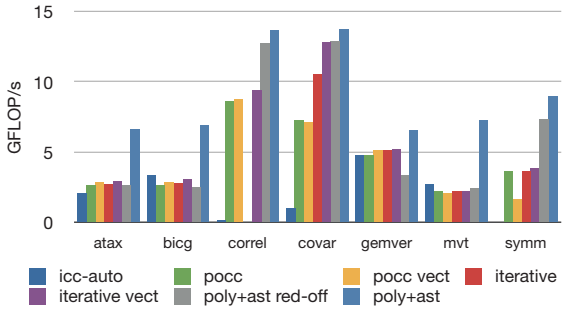

Fig. 9: PolyBench on 8-core Nehalem (contains reduction parallelism)

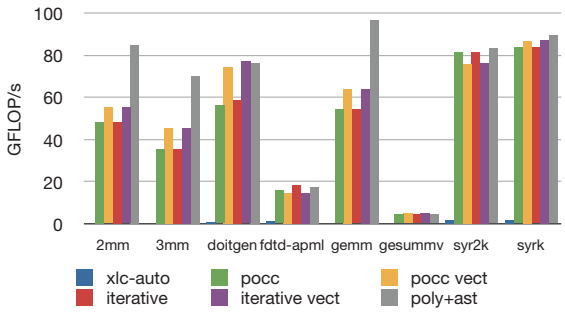

Fig. 10: PolyBench on 32-core Power7 (doall parallelism is dominant)

Figure 8 shows the performance numbers of benchmarks whose major source of parallelism is pipeline. There is an additional variant, *poly+ast pipe-off*, which shows the performance using wavefront doall as with pocc variants, instead of pipeline parallelism by runtime supports (Sec. IV-D). The variants of

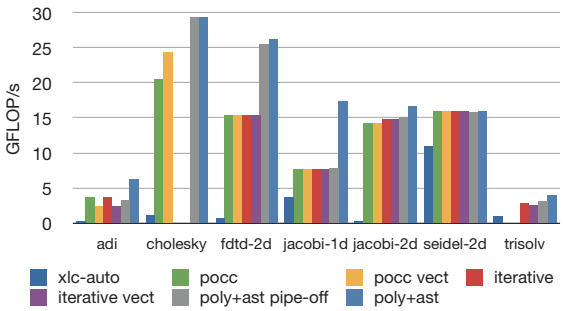

Fig. 11: PolyBench on 32-core Power7 (pipeline parallelism is dominant)

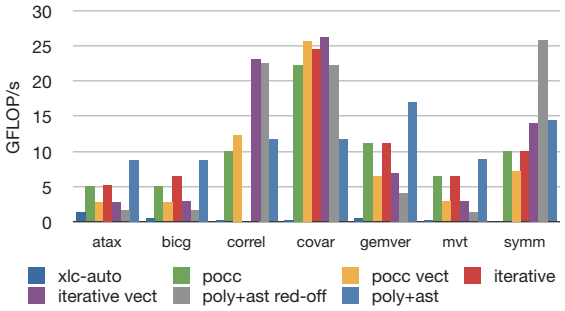

Fig. 12: PolyBench on 32-core Power7 (contains reduction parallelism)

the pocc framework, *pocc*, *pocc vect*, *iterative* and *iterative vect*, have similar performance except for cholesky while *poly+ast pipe-off* shows higher performance for adi, cholesky, and fdtd-2d due to better selections of transformations such as permutation and retiming. Further, the runtime supports for pipeline parallelism brought additional improvements for most benchmarks. The details of performance improvement by the pipeline and reduction runtime are summarized in Table III.

Figure 9 show the performance numbers of benchmarks that include reduction parallelism. The variant *poly+ast red-off* stands for the version that uses inner doall parallelism instead of reduction parallelism at the outermost. Due to the finer-grained parallelism, *poly+ast red-off* is slower than the pocc variants for atax, bicg, and gemver while *poly+ast* is the winner for all benchmarks. Because of the reduction parallelism, *poly+ast* has larger flexibility for loop orders and selected loop permutations with outermost parallelism and good data locality.

C. Performance Results on Power7 with 32 cores

The performance in GFLOP/s on *Power7* using 32 cores are shown in Figures 10, 11, 12, each of which corresponds to the benchmarks with doall, pipeline and reduction parallelism. On *Power7*, *poly+ast* shows much higher performance than both of iterative and model-driven approaches of pocc even for the benchmarks without pipeline nor reduction parallelism. This could stem largely from the large L3 cache (32MB) and TLB (32MB) of *Power7*, where inter-tile loop orders can have larger impact on the performance compared with *Nehalem*. As with the results in Sec. V-B, Figures 11 and 12 show the additional benefits for *poly+ast* thanks to the pipeline and reduction runtimes. However, the reduction support did not always bring performance improvement; *poly+ast red-off* shows better performance than *poly+ast* for correl, covar and covar. A challenge for efficient reduction is the aggregation of partial results into the final result. The current runtime only support sequential aggregation across all threads and thereby can be scalability bottleneck for large size of arrays. Further, this bottleneck could be more critical on *Power7* with larger number of cores than *Nehalem*. This issue of sequential aggregation will be investigated in future work.

D. Performance Breakdown

Table IV show the performance breakdown of the *poly+ast* approach, which shows the performance in GFLOP/s when turning on specific functionalities in the proposed framework. The column *fuse* represents the performance when only loop fusion is turned on, *fuse + pm* is the performance with fusion and permutation, *tiling* is the case where only loop tiling is applied, and *all* is the full functionality of *poly+ast*. It shows that selecting good permutation order is critical for 2mm and 2mm, while tiling brings certain performance gain even without permutation/fusion. The combination of all functionality always shows the best performance for these benchmarks.

	Nehalem 8-core			Power7 32-core		
	2mm	3mm	jacobi-2d	2mm	3mm	jacobi-2d
fuse	2.23	1.82	9.04	1.09	0.76	5.80
fuse+pm	7.25	6.24	9.04	19.22	16.73	5.80
tiling	10.72	8.65	7.68	10.49	8.44	5.03
all	19.09	14.14	10.92	62.63	53.57	6.98

TABLE IV: Breakdown of poly+ast approach (GFLOP/s)

VI. Related Work

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

AST-based transformation frameworks have a long history [1], [2]. There are a lot of pioneering works for parallelizing and locality optimizing compilers in both research and industry [3], [4], [5], [6], [7]. Loop fusion heuristics were initially designed as locality-enhancing optimizations, in isolation from other loop nest transformations [8], [9], [10], [11]. These non-polyhedral approaches are restricted in their ability to model the interplay of loop fusion with equally important optimizations such as loop tiling. The lack of a powerful representation for dependences and composition of transformations also restricted the study of enabling loop transformations to enhance the applicability of loop fusion.

Several heuristics for loop fusion combined with tiling have been proposed [12], [13], but do not capture the interplay between loop transformations, back-end optimizations performed by the compiler, and components of the target architecture. Megiddo and Sarkar [10] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed.

Recent research on integrating fusion and tiling in a single heuristic based on the polyhedral model led to the Pluto framework by Bondhugula et al. [14], [15]. Pluto [28] is a polyhedral framework for locality and parallelism optimizations, which handles the whole loop transformations by solving ILP formulation based on dependence distances. It inherits the flexibility of the tiling hyperplane method [33], [34] to build complex sequences of enabling and communication-minimizing transformations, subsuming most compositions of loop transformations into a single optimization step. It does identify good parallelism-locality trade-offs using a target-independent cost model. However it suffers several key deficiencies in its effort to integrate all performance objectives, as illustrated in Sec. II and in our experiments. Pouchet et al. used empirical search in order to find better fusion/code motion transformations [26], [24], whose performance is reported as pocc-iterative in our experiments. While it copes with the fusion cost model limitations, this approach still suffers from the data locality minimization objective inside the chosen outer-loop fusion, and performance can be greatly improved in numerous cases, as shown in our experiments. More recent approaches to the selection of loop fusion in the polyhedral model have been proposed, such as a cost model for loop fusion based on the prefetch streams (especially IBM Power chips) [35]. The R-Stream compiler integrates a cost model for joint fusion and parallelism [36]. It balances locality optimizations with doall parallelism while our approach first optimizes data locality in the polyhedral phase and doall/reduction/pipeline parallelism is detected in the latter AST-based phase. The smartfuse fusion heuristic of Pluto (corresponding to the pocc numbers in our experiments) has also recently been studied and refined [37]. As with the empirical search approach [26], [24], this also focuses on the fusion cost model while the DL-based cost model in our approach guides fusion and permutation in an integrated manner.

As an integration approach, the CHiLL scripting system [38] combines polyhedral and AST transformations. Note that CHiLL expects the user to guide transformations while our approach is fully automated by compiler heuristics.

VII. Conclusion

The complexity of modern multi-core processors has significantly increased the expectations of optimizing compilers: to achieve good performance we expect such compilers to perform program transformations addressing all major performance anomalies, such as data locality, coarse-grain parallelism and also fine-grain / SIMD parallelism. Previous approaches have demonstrated the benefit of using the polyhedral compilation framework to design an integrated optimization problem that attempts to address all performance objectives in a single stage.

In this paper we have shown that the performance of a state-of-the-art integrated polyhedral approach could be vastly improved. We have shown that a decoupled optimization approach allowed to simplify the constraints on the polyhedral transformation stage while using more precise cost models, leading to simpler and more effective codes to be generated. Furthermore, we have shown that several key transformations such as inter-tile parallelization, when performed using a syntactic/AST framework, significantly improved performance over conventional polyhedral frameworks. We demonstrated strong performance improvement over both a state-of-the-art integrated polyhedral compiler and an iterative compilation framework that searches across certain loop fusion structures. Our framework achieves performance improvements of up to 3× on PolyBench/C benchmark suite executed on Intel Nehalem and IBM Power7 processors.

ACKNOWLEDGMENT

This work for was supported in part by the Center for Domain Specific Computing (NSF Expeditions in Computing Award CCF-0926127). We also gratefully acknowledge the developers of the polyhedral and AST compiler infrastructure components used in this work.

REFERENCES

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

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

[3] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the perfect benchmarks," *IEEE Trans. on parallel and distributed systems*, vol. 9, no. 1, Jan. 1998.

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

[5] V. Sarkar, "Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers," *IBM J. Res. & Dev.*, vol. 41, no. 3, 1997.

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

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

[8] K. Kennedy and K. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing*, 1993.

[9] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 424–453, 1996.

[10] N. Megiddo and V. Sarkar, "Optimal weighted loop fusion for parallel programs," in *symposium on Parallel Algorithms and Architectures*, 1997.

[11] S. Singhai and K. McKinley, "A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality," *The Computer Journal*, vol. 40, no. 6, pp. 340–355, 1997.

[12] M. Wolf, D. Maydan, and D.-K. Chen, "Combining loop transformations considering caches and scheduling," in *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, 1996.

[13] A. Qasem and K. Kennedy, "Profitable loop fusion and tiling using model-driven empirical search," in *Proc. of the 20th Intl. Conf. on Supercomputing (ICS'06)*, 2006.

[14] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International conference on Compiler Construction (ETAPS CC)*, 2008.

[15] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[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)*.

[17] "Polybench/c 3.2," *http://sourceforge.net/projects/polybench/*.

[18] "the polyhedral compiler collection," *http://www.cs.ucla.edu/ pouchet/-software/pocc/*.

[19] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing doacross loop dependencies in openmp," in *9th International Workshop on OpenMP (IWOMP)*, 2011.

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

[21] ——, "Some efficient solutions to the affine scheduling problem, part II: multidimensional time," *Intl. J. of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec. 1992.

[22] 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," *Intl. J. of Parallel Programming*, vol. 34, no. 3, 2006.

[23] P. Feautrier, "Dataflow analysis of scalar and array references," *Intl. J. of Parallel Programming*, vol. 20, no. 1, pp. 23–53, Feb. 1991.

[24] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop transformations: Convexity, pruning and optimization," in *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*.

[25] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, 2004.

[26] 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)*.

[27] J. Ferrante, V. Sarkar, and W. Thrash, "On Estimating and Enhancing Cache Effectiveness," *Proc. LCPC 91*, vol. 589, 1991.

[28] 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.

[29] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, "Analytical bounds for optimal tile size selection," in *ETAPS International Conference on Compiler Construction (CC'12)*.

[30] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008.

[31] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile reduction: The first step towards tile aware parallelization in openmp," in *9th International Workshop on OpenMP (IWOMP)*, 2011.

[32] T. P. compiler project, "http://pace.rice.edu."

[33] F. Irigoin and R. Triolet, "Supernode partitioning," in *ACM SIGPLAN Principles of Programming Languages*, 1988.

[34] M. Griebl, P. Faber, and C. Lengauer, "Space-time mapping and tiling – a helpful combination," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 3, pp. 221–246, Mar. 2004.

[35] 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*, ser. PACT '10, 2010.

[36] "R-stream high-level compiler," *http://www.reservoir.com*.

[37] S. Mehta, P.-H. Lin, and P.-C. Yew, "Revisiting loop fusion in the polyhedral framework," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14, 2014.

[38] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," 2008.