Loopy: Programmable and Formally Verified Loop Transformations

Kedar S. Namjoshi[1] and Nimit Singhania[2]

[1] Bell Laboratories, Nokia kedar@research.bell-labs.com
[2] University of Pennsylvania nimits@seas.upenn.edu

Abstract. This paper presents a system, Loopy, for programming loop transformations. Manual loop transformation can be tedious and error-prone, while fully automated methods do not guarantee improvements. Loopy takes a middle path: a programmer specifies a loop transformation at a high level, which is then carried out automatically by Loopy, and formally verified to guard against specification and implementation mistakes. Loopy's notation offers considerable flexibility with assembling transformations, while automation and checking prevent errors. Loopy is implemented for the LLVM framework, building on a polyhedral compilation library. Experiments show substantial improvements over fully automated loop transformations, using simple and direct specifications.

1 Introduction

Restructuring loops in programs to match a target architecture can yield dramatic improvements in run-time, e.g., by exploiting parallelism, by matching the pattern of memory accesses to cache sizes and memory layout, and through judicious placement of prefetching instructions. Re-structuring a loop is, however, a difficult task to do manually – it is tedious and error-prone, and the resulting code is difficult to understand and to maintain. This problem has long been recognized, and it has led to decades of work on sophisticated algorithms which automatically optimize loops. An *optimal* transformation algorithm remains out of reach, though, as the underlying optimization questions are computationally difficult, having to balance a number of potentially conflicting objectives such as code size, parallelism, and cache locality. Performance gains are, therefore, quite variable, even with state-of-the-art compilers.

A remedy to these problems is a semi-automatic optimization approach, where the work is split between the programmer and the compiler: the transformation is *specified* at a high level by the programmer, but the task of *implementing* it falls to the compiler and is done fully automatically. We explore this approach in our work. The approach has multiple benefits. In comparison with manual optimization, first, the programmer is relieved from the manual effort of rewriting the program and the possibility of introducing errors. Second, the specification or parts of it can be re-used, resulting in a library of transformations which further reduces manual effort. Third, the specification acts as a certificate for the transformation, it can be verified during application by the compiler, and even checked independently as a further guarantee of correctness.

In comparison with automatic optimization, significant performance gains can be achieved using fairly simple specifications, as we observe in our experiments, and has also been observed previously [6,12,9]. An alternate approach often used to optimize programs is via calls to hand-optimized libraries like Intel's MKL [13] and ATLAS [27]. A semi-automatic optimization approach can complement these approaches and be useful when they do not produce the desired performance improvement.

An effective semi-automatic optimization framework must (1) provide source-code level semantics for the specifications, so that a programmer can easily describe and reason about a transformation, and (2) automatically check the correctness of a specification, so that incorrect transformations are not implemented. Most existing works fall short on at least one of these. CHiLL [6,22], POET [28], Orio [12] and X Language [7] allow program-level transformations, but do not check correctness. On the other hand, URUK [9] verifies transformations, but the specification is on the underlying mathematical representation, which can be too abstract for a programmer. Our work ensures both, by exposing a small set of basic operators, which can be combined together to form complex transformations, and by associating a formal declarative semantics to each operator, which is used for automation and checking.

A transformation in our language is specified as a composition of (instances of) primitive, building-block operations. The language supports four basic operations, which allow arbitrary affine transformations on loop nests and flexible ways of splitting and merging loops. Programmers label loops and sections of loops with loop tags, which are used as handles to describe focused piece-wise transformations on the loop components. The combination of powerful primitive operations with piece-wise application makes it easy for a programmer to assemble a complex loop transformation.

We define formal semantics for each of the basic operations via polyhedral representation of programs. These semantics are used to implement the transformations on the source program, and to ensure that the transformed program is semantically equivalent to the original program.

We have implemented a prototype tool, Loopy[3], building on a polyhedral library for the LLVM compiler framework. Loopy is essentially an interpreter for the specification language, it carries out the specified transformation and verifies its correctness. We have evaluated Loopy on Polybench, a benchmark suite for polyhedral model based tools. The experiments show significant improvements in performance over fully automated methods, with simple specifications.

[3] The name is an obvious pun on 'loop' transformation. Moreover, "loopy" is slang for "crazy", which we hope is **not** how this work strikes the reader!

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  {
    Init: C[i][j] = 0;
  }
  for(k=0;k<N;k++){
   Mult:
    C[i][j] += A[i][k]*B[k][j];
  }
 }
}
```

(a) C++ matrix multiplication fragment

```
for(i=1;i<N;i++){
 Div:
  A[i] = A[i]/A[i-1];
}
```

(b) Correctness Checking

Fig. 1: Illustrative Programs to compare Loopy with URUK and CHiLL

2 Illustrative Example

We use simple C/C++ programs, shown in Figure 1, to illustrate capabilities of Loopy as compared to two other semi-automatic optimization approaches, CHiLL [6] and URUK [9]. CHiLL is representative of systems that specify transformations at source program level, however do not provide any correctness guarantees. URUK is representative of the systems that provide correctness guarantees but operate on alternate representations of programs.

We first focus on improving cache usage on a single-core machine for a matrix multiplication program, shown in Figure 1a. A multidimensional array in a C/C++ program is stored in row-major order and thus, accesses to elements of array B exhibit poor spatial locality as consecutive accesses occur along the column. Also, accesses to both A and B exhibit poor temporal locality, as each element is accessed repeatedly in different iterations of the outer loops. We would like to reorder these accesses to improve locality. A sequence of loop transformations that would achieve this is: the loop is first split into its component sections, `Init` and `Mult`, the loop indices j and k are interchanged in `Mult` and finally, the resulting loop is tiled. The first transformation enables the subsequent operations on `Mult`, the second operation improves spacial locality for B and the last operation improves temporal locality for arrays A and B.

We present optimization scripts in Loopy and URUK to implement these loop transformations. Figure 2a shows the Loopy script. The first operation, *realign*, readjusts the number of common loops between two adjacent loop components. Here, the number of common loops is set to 0, which results in complete splitting of the loops. The second one, *affine*, applies the specified affine transformation to the iterators of a loop component. As is well known, affine transformations can be used to represent common transformations such as loop permutations, loop reversal, loop shifting, loop scaling, and loop tiling. For this example, the affine transformations permute loop indices and tile the loop.

```
realign(Init, Mult, 0)
affine(Mult, {[i,j,k] -> [i,k,j]})
affine(Mult, {[i,j,k] -> [i1,j1,k1,i2,j2,k2]:
        i1 = [i/64] and i2 = i%64 and
        j1 = [j/64] and j2 = j%64 and
        k1 = [k/64] and k2 = k%64})
```

(a) Loopy script

```
FISSION([0], [0], [1])
INTERCHANGE([1, 0])
TILE([1, 0], 64, 64)
STRIPMINE([1], 64)
INTERCHANGE([1, 0])
INTERCHANGE([1, 0, 0])
```

(b) URUK script

Fig. 2: Optimization scripts for program in Fig. 1a.

In URUK, the transformations are applied on a polyhedral representation of programs, where each statement is represented by a vector of integers (we refer to this representation in more detail in Section 3.2). Figure 2b shows a script that is input to URUK. The `FISSION` operation splits the loop into component sections. The first `INTERCHANGE` operation interchanges loop indices. The remaining operations implement tiling. The `TILE` operation tiles the two inner loops, while `STRIPMINE` sections the outermost loop. The `INTERCHANGE` operations are used to permute loop indices in correct order. In our view, the script for Loopy is simpler and more direct than that for URUK. Loop tags `Init` and `Mult` make it possible to focus the transformations on specific parts of the loop, without referring to cryptic representations of statements.

We now use a sample loop, shown in Figure 1b, to illustrate requirements for verifying correctness of transformations. It is easy to see that reversing the loop `Div` is incorrect. Suppose the loop is executed with A initialized to $[1, 2, 3, 4]$. On executing the loop, A is updated to $[1, 2, 3/2, 8/3]$. However, on executing the reverse loop, A is updated to a different array $[1, 2, 3/2, 4/3]$. Hence, a transformation that reverses this loop is incorrect. Our experiment with CHiLL shows that it implements this transformation without complaint. Loopy, on the other hand, does not carry out the transformation, it generates a violation that points to the execution order dependency that is not preserved. As mistakes such as this are easy to make when complex transformations are involved, correctness checking is essential.

3 Design

We describe the design of Loopy. The tool represents a point in the trade-off between manual optimization (much effort for good performance) and automatic optimization (little manual effort but variable performance).

Loopy may be viewed as a compiler extension that reads in and interprets a transformation script. A script is a sequence of (instances of) primitive building-block transformations. Loopy's interpreter carries out the transformation defined by the composition of the sequence of operations in the script. Loopy contains a verifier, which checks the script and informs the user of an error if carrying it out may result in a program with differing semantics.

In more detail, the transformation process works as follows. Initially, the polyhedral model (parameterized statements, iteration domains, initial schedules and dependency maps) is constructed from the program. Loop components and their domains are identified and computed from the loop tags in the program. Following that step, the transformation script is read in, one operation at a time, and interpreted. The interpretation of each operation results in a new schedule and (possibly) the update of existing loop components and the creation of new loop components. After all of the transformations are completed, the final program schedule is checked by the verifier against the execution order dependency maps, and any dependency violations are reported to the programmer.

This section presents the specification language (Section 3.1) with the formal semantics for the building blocks (Section 3.2). Verification of transformations is described in Section 4.

3.1 Specification Language

We give an overview of the operations available to a programmer to identify loop components and specify transformations.

Loop Tagging Users may label loops or loop sections with tags and use those as handles in a script. With tags, transformations can be focused on specific portions of a loop. We re-use the idea of scope blocks from C and C++, and define a block of code surrounded by curly braces ({ }) as a *loop component*. We also assign a label to the first statement of the component and use it as the loop tag and a handle in the optimization script. Labels `Init` and `Mult` in Figure 1a are examples of loop tags. This re-purposing of scope blocks and labels avoids the need for modifications to the source language.

Building-block Operations Next, we define the primitive operators currently supported in Loopy. (We expect to add more as we gain experience with the tool.) The operators are illustrated by the examples in Figure 3.

realign. The realign operator, written as $\mathsf{realign}(l_1, l_2, n)$, is used to split or merge two adjacent loop components l_1 and l_2. This operator realigns the loop components so that the number of common loops in the loop nests of l_1 and l_2 is n. Consider the program in Figure 3a which consists of loops $L1$ and $L2$ with indices i and j sharing one common loop. The result of applying the operators $\mathsf{realign}(L1, L2, 0)$ and $\mathsf{realign}(L1, L2, 2)$ are shown in Figures 3b and 3c respectively. The first splits the components into independent loop nests, while the second merges them into a single loop nest.

lift. The lift operator, written as $l_2 = \mathsf{lift}(l_1, n)$, returns a handle l_2 to the nth level loop in the loop nest of l_1. It does not transform the program schedule. The ability to select a sub-loop is useful when a subsequent operator is to be applied to an outer loop of the component. For example, applying the operator

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  ....
       ....
 }
 for(j=0;j<N;j++){
  L2:  ....
       ....
 }
}
```

(a) P

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  ....
       ....
 }
}
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L2:  ....
       ....
 }
}
```

(b) Q

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  ....
       ....
  L2:  ....
       ....
 }
}
```

(c) R

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L:  ....
      ....
      ....
      ....
 }
}
```

(d) S

```
for(i=0;i<N1;i+=2){
 for(j=i/2;j<N2;j++){
  L:  ....
      ....
      ....
      ....
 }
}
```

(e) T

```
for(i=0;i<N;i++){
 for(j=0;j<N/2;j++){
  L3:  ....
       ....
 }
 for(j=N/2;j<N;j++){
  L4:  ....
       ....
 }
}
```

(f) U

Fig. 3: Examples of basic operations: (a) P: original program (b) Q: $\mathsf{realign}(L1, L2, 0)$ on P (c) R: $\mathsf{realign}(L1, L2, 2)$ on P (d) S: $L = \mathsf{lift}(L1, 2)$ on R (e) T: $\mathsf{affine}(L, \{[i, j] \to [2i, i+j]\})$ on S (where, $N1 = 2N$ and $N2 = N + i/2$) (f) U: $(L3, L4) = \mathsf{isplit}(L, \{[i, j] : j < N/2\}, 1)$ on S.

$L = \mathsf{lift}(L1, 2)$ to the program in Figure 3c returns handle L to the merged loop as shown in Figure 3d, which is used for further transformations.

affine. The affine operator, written as $\mathsf{affine}(l, f)$, is used to restructure the loop nest of l according to a general affine function f. A new set of iterators is defined, and each new iterator is assigned an affine function of old iterators, which is given by the function f. As is well known, affine functions can be used to represent a number of useful transformations, such as loop reversal (with $\{[i, j] \to [-i, j]\}$), loop permutation (with $\{[i, j, k] \to [j, k, i]\}$), loop scaling and shifting (with $\{[i, j] \to [2i + 1, j]\}$) and loop rotation (with $\{[i, j] \to [i + j, i - j]\}$). Further, we also allow integer division by a constant which further enables loop tiling (as in

$\{[i,j] \rightarrow [i/32, j/32, i\%32, j\%32]\}$). Figure 3e shows an affine transformation of program S in Figure 3d.

isplit. The isplit operator, written as $(l_1, l_2) = \mathsf{isplit}(l, p, n)$, is used to split the index set of the loop nest of l to create two new loop nests given by handles l_1 and l_2, such that the indices in l_1 satisfy the predicate p and those in l_2 satisfy $\neg p$. The value n specifies the loop nest level at which the splitting occurs or the number of common loops after the splitting. For example, consider the operator $(L3, L4) = \mathsf{isplit}(L, \{[i,j] : j < N/2\}, 1)$ on program S in Figure 3d. The result is shown in Figure 3f, where $L3$ consists of iterations of j with values less than $N/2$ and $L4$ consists of those with values greater than $N/2$.

3.2 Formal Semantics

We use the polyhedral model of programs to define the semantics of these operators. The polyhedral model represents a program as a collection of statements each of which is parameterized by the iterators of its enclosing loop nest. This model is easy to manipulate via algorithms and hence, is widely used in automated analysis and optimization tools. It allows a rich class of transformations, including the ones expressed in our specification language. We briefly describe the model and follow that with a precise formulation of each operator.

Polyhedral model The polyhedral model represents a program as a collection of statement instances, S, defined over an *iteration space* formed by the possible values of iteration variables. A statement instance has the form $s[i_1, i_2, \ldots, i_M]$, where s is a statement in the program, and $\boldsymbol{i} = (i_1, \ldots, i_M)$ is a vector of integer variables representing the iterators of the loop nest which contains s. The set of valid instances of statement s is given by constraining $\boldsymbol{i}$, in terms of affine inequalities on the iterators and the external parameters of the program; the set of valid vectors is called the *iteration domain* of the statement.

A partial *schedule* function, denoted θ, maps a statement and a point in the iteration space to a linearly ordered (abstract) "time" domain. The time value given by $\theta(s, \boldsymbol{i})$ is referred to as the *schedule vector* of instance $s[\boldsymbol{i}]$. We use a schedule vector of the following shape: for statement instance $s[\boldsymbol{i}]$,

$$\theta(s, \boldsymbol{i}) = (p_0, j_1, p_1, \ldots, j_M, p_M)$$

The p-entries represent positions, the j-entries the iteration number, as explained next. For convenience, we sometimes separate the position and iteration entries in a vector, representing this as the pair of vectors $(\boldsymbol{p}, \boldsymbol{j})$.

Consider the abstract syntax tree of a loop nest. Each node represents a basic program statement or a loop; its children (if any) represent nested substatements of that statement, and are ordered by their sequence in the program. The natural number p_0 identifies a top-level loop or statement; the number p_1 identifies one of the sub-statements of the p_0'th statement. Thus, the sequence of numbers $p_0, p_1, p_2, \ldots$ fixes a path from root to leaf in the abstract syntax tree of the outermost loop. The integer j_{k+1} fixes a particular iteration of the loop statement at position $p_0, p_1, \ldots p_k$.

To simplify manipulation, we fix the dimension of the schedule vector to the length of the longest root-to-leaf path in the abstract syntax tree, filling missing entries with 0's. As a convenient notation, let $pos_\theta(s, \boldsymbol{i})$ denote the position vector $(p_0, p_1, \ldots, p_M)$, and let $it_\theta(s, \boldsymbol{i})$ denote the iteration vector $(j_1, j_2, \ldots, j_M)$ in the schedule vector $\theta(s, \boldsymbol{i})$.

Let $\boldsymbol{u} = \theta(s, \boldsymbol{i})$ and $\boldsymbol{v} = \theta(t, \boldsymbol{j})$ be schedule vectors for statement instances $s[\boldsymbol{i}]$ and $t[\boldsymbol{j}]$ in the current schedule θ. If $\boldsymbol{u}$ is lexicographically smaller than $\boldsymbol{v}$ i.e. $\boldsymbol{u} \prec \boldsymbol{v}$ then $s[\boldsymbol{i}]$ is executed before $t[\boldsymbol{j}]$. Every loop transformation alters only the schedule θ; the original set of statement instances and iteration domains remains unchanged. I.e., only the execution order of the same set of statement instances is rearranged.

As an example, consider the program in Figure 1a. The program consists of statements $s_{Init}[i, j]$ and $s_{Mult}[i, j, k]$. The iteration domains of these statements are $0 \leq i < N \wedge 0 \leq j < N$ and $0 \leq i < N \wedge 0 \leq j < N \wedge 0 \leq k < N$ and their initial schedule is $(0, i, 0, j, 0, 0, 0)$ and $(0, i, 0, j, 1, k, 0)$ respectively. The corresponding position vectors in the initial schedule are $[0, 0, 0, 0]$ and $[0, 0, 1, 0]$. The execution order of the program is implicitly captured in the schedule vectors: for an iteration $[i_0, j_0]$, $s_{Init}[i_0, j_0]$ is executed before $s_{Mult}[i_0, j_0, k]$ since, $[0, 0, 0, 0] \prec [0, 0, 1, 0]$ and similarly, for two iterations $[i_0, j_0]$ and $[i_1, j_1]$ if $[i_0, j_0] \prec [i_1, j_1]$, then $s_{Init}[i_0, j_0]$ is executed before $s_{Init}[i_1, j_1]$.

Consider program A in Figure 4a. Each statement is annotated with its position vector. For example, statement s_4 is annotated with vector $[1, 1, 0]$, as the outermost loop with iterator i is at position 1, the second loop with iterator j is again at position 1, while the statement itself is at position 0.

Loop Component Let $\mathcal{L}$ denote the set of loop components in the program. Let $P(l)$ be the set of positions represented by component l. I.e., $P(l)$ is the set of positions of statements in the subtree rooted at l. Given P and schedule θ, let $D_\theta(l)$ denote the set of statement instances contained in the scope block of the component. I.e., $s[\boldsymbol{i}]$ is in $D_\theta(l)$ if and only if its position vector $pos_\theta(s, \boldsymbol{i})$ is in $P(l)$. In Figure 4a, for loop component $L1$, $P(L1) = \{[0, 0, 0], [0, 0, 1]\}$, and $D_\theta(L1) = \{s_0[i, j], s_1[i, j]\}$ for the implied schedule θ.

Basic Transformations Now, we define the semantics of each basic operation. This is defined as a change of the schedule of the program. I.e., a basic transformation only rearranges the execution order of statements, it does not change the set of statements. In each case, we specify the semantics of an operation as a map from a current schedule θ to the new schedule θ'.

realign. In a $\mathsf{realign}(l_1, l_2, n)$ operation, the position vector of statements in loop component l_2 is updated so that, statements in l_1 and l_2 share the first n loops, and l_2 is next to l_1 in the nth loop after the update. To define this, we first compute the current gap between the positions of statements in l_1 and l_2. Let

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  <s0>[0,0,0]
       <s1>[0,0,1]
 }
}
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L2:  <s2>[1,0,0]
       <s3>[1,0,1]
 }
 for(j=0;j<N;j++){
  L3:  <s4>[1,1,0]
       <s5>[1,1,1]
 }
}
```

(a) A

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  <s0>[0,0,0]
       <s1>[0,0,1]
  L2:  <s2>[0,0,2]
       <s3>[0,0,3]
 }
 for(j=0;j<N;j++){
  L3:  <s4>[0,1,2]
       <s5>[0,1,3]
 }
}
```

(b) B

```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  L1:  <s0>[0,0,0,0]
       <s1>[0,0,1,0]
 }
}
for(i=0;i<N;i++){
 for(j=0;j<N1;j++){
  for(k=0;k<N2;k++){
   L2:  <s2>[1,0,0,0]
        <s3>[1,0,0,1]
  }
 }
 for(j=0;j<N;j++){
  L3:  <s4>[1,1,0,0]
       <s5>[1,1,1,0]
 }
}
```

(c) C

Fig. 4: Examples illustrating semantics of basic operations (a) A: original program with statements labeled with position vectors (b) B: realign$(L1, L2, 2)$ on A (c) C: affine$(L2, \{[i, j] \to [i, j/32, j\%32]\})$ on A (where $N1 = N/32 + 1, N2 = \min\{32, N - j * 32\}$). Note the change in position vectors in programs B and C.

the lexicographically largest position in l_1 be $\boldsymbol{p}_1$ i.e. $\boldsymbol{p}_1 = max(\boldsymbol{q}), \boldsymbol{q} \in P(l_1)$ and the lexicographically smallest position in l_2 be $\boldsymbol{p}_2$ i.e. $\boldsymbol{p}_2 = min(\boldsymbol{q}), \boldsymbol{q} \in P(l_2)$ and let gap, $\boldsymbol{g}$ be their difference, i.e. $\boldsymbol{g} = \boldsymbol{p}_2 - \boldsymbol{p}_1$. For example, in Figure 4a, the largest position in $L1$ is $[0, 0, 1]$ and the smallest position in $L2$ is $[1, 0, 0]$ and their gap is $[1, 0, -1]$. Now, to get the desired update, this gap should be removed, and thus, $\boldsymbol{g}$ must be subtracted from position vectors of statements in l_2. To position statements in l_2 next to those in l_1 at the nth loop, the nth value in their position vectors must be incremented by 1. Therefore, for a statement $s[\boldsymbol{i}] \in D(l_2)$, the new position vector, $pos_{\theta'}(s, \boldsymbol{i})$ is $pos_\theta(s, \boldsymbol{i}) - \boldsymbol{g} + \boldsymbol{I}(n)$, where $\boldsymbol{I}(n)$ is an indicator vector with component $I_j(n) = 1$ if $j = n$ and 0 otherwise.

In a realign operation, the positions of statements with positions *after* those in l_2 must also be updated by the same amount as statements in l_2. This is because, the relative difference between the positions of statements in l_2 and those after l_2 must remain same. For example, consider the operation realign$(L1, L2, 2)$ on program A in Figure 4a. The resulting program is shown in Figure 4b. To preserve the relative position with respect to $L2$, the statements in $L3$ must also be moved inside the loop with iterator i.

The overall transformation is:

$$\phi_{\mathsf{realign}}(\boldsymbol{p}, \boldsymbol{j}) = \begin{cases} (\boldsymbol{p} + \boldsymbol{p_1} - \boldsymbol{p_2} + \boldsymbol{I}(n), \boldsymbol{j}) & \boldsymbol{p} \succeq \boldsymbol{p_2} \\ (\boldsymbol{p}, \boldsymbol{j}) & \text{otherwise} \end{cases}$$

Note that, $\boldsymbol{p} \succeq \boldsymbol{p_2}$ denotes all position vectors lexicographically greater than or equal to p_2 and thus, represents all statements in l_2 or those after l_2 in the syntax tree. Now, the new schedule $\theta' = \phi_{\mathsf{realign}} \circ \theta$ and for each component $l \in \mathcal{L}$, $P'(l) = \phi_{\mathsf{realign}}(P(l))$.

lift. The operation $l_2 = \mathsf{lift}(l_1, n)$ is different from the others in that it is only definitional, it does not alter schedules. This operation adds a new component l_2 to $\mathcal{L}$. This component consists of all statements within the nth outer loop of l_1. These are precisely the statements in the program for which the position vectors match those in $P(l_1)$ on the first n values. Hence, the operation returns a loop component l_2 such that,

$$P(l_2) = \{\boldsymbol{q} : (q_0, q_1, \ldots, q_{n-1}) \equiv (p_0, p_1, \ldots, p_{n-1}) \text{ for some } \boldsymbol{p} \in P(l_1)\}$$

affine. The operation $\mathsf{affine}(l, f)$ applies function f to the iteration vector of statements in l. As f may increase the number of iterators, it may be required to update the position vectors of statements as well. Let f take a iterators and generate b iterators as output, i.e. $f : Z^a \rightarrow Z^b, b \geq a$. First, the number of dimensions of position vectors must be increased by $b - a$ dimensions to account for the additional iterators. Next, the dimensions of old position vectors must be mapped to the new set of dimensions, such that the relative position difference between statements in l and those outside l remains the same and also, the statements in l are contained inside the new iteration set.

The position of statements outside l remains the same with only zeros padded in the end. However, the position vectors of statements within l is updated as follows. To preserve the relative position difference, we map the first a dimensions of old position vector to the a dimensions of the new one. To contain the statements inside the new iteration set, every subsequent dimension is shifted to the right by $b - a$ positions and thus, kth dimension of old position vector is mapped to $(k + b - a)$th dimension in the new position vector. The remaining dimensions are filled with zeros. For example, Figure 4c shows the result of operation $\mathsf{affine}(L2, \{[i, j] \rightarrow [i, j/32, j\%32]\})$ on program in Figure 4a. As can be seen, for statements in $L1$ and $L3$, a zero is added in the last dimension, while for $L2$, zero is added in the third dimension of the position vector.

The overall transformation is:

$$\phi_{\mathsf{affine}}(\boldsymbol{p}, \boldsymbol{j}) = \begin{cases} ((p_0, \ldots, p_a, \overbrace{0, \ldots, 0}^{\text{(b-a)}}, p_{a+1}, \ldots, p_M), f(\boldsymbol{j})) & \boldsymbol{p} \in P(l) \\ ((\boldsymbol{p}, \overbrace{0, \ldots, 0}^{\text{(b-a)}}), (\boldsymbol{j}, \overbrace{0, \ldots, 0}^{\text{(b-a)}})) & \text{otherwise} \end{cases}$$

As in the case of realign, $\theta' = \phi_{\mathsf{affine}} \circ \theta$ and for each component $l \in \mathcal{L}$, $P'(l) = \phi_{\mathsf{affine}}(P(l))$.

isplit. The operation $(l_1, l_2) = \mathsf{isplit}(l, pred, n)$ splits the iteration domain of the loop component l into two new components, l_1 and l_2. Component l_1 consists of statements in l whose current iteration vector satisfies the predicate $pred$, i.e. $pred(it_\theta(s, \boldsymbol{i})) = true$, while l_2 consists of the remaining statements. Note that the first component l_1 takes the place of l, while l_2 is inserted after l_1 in the nth loop. To accommodate this, statements in l_2 and all subsequent statements must be moved down by one position. This is done just as in the case of realign operation. Let the lexicographically smallest schedule vector in $P(l)$ be $\boldsymbol{p_m}$. The transformation here is

$$\phi_{\mathsf{isplit}}(\boldsymbol{p}, \boldsymbol{j}) = \begin{cases} (\boldsymbol{p}, \boldsymbol{j}) & \boldsymbol{p} \prec \boldsymbol{p_m} \vee \boldsymbol{p} \in P(l) \wedge pred(\boldsymbol{j}) \\ (\boldsymbol{p} + \boldsymbol{I}(n), \boldsymbol{j}) & \text{otherwise} \end{cases}$$

As in the previous cases, $\theta' = \phi_{\mathsf{isplit}} \circ \theta$ and for each component $l \in \mathcal{L}$, $P'(l) = \phi_{\mathsf{isplit}}(P(l))$. Further, the new components l_1 and l_2 are defined as follows:

$$P'(l_1) = \{(\boldsymbol{p}) : \boldsymbol{p} \in P(l)\}$$

$$P'(l_2) = \{(\boldsymbol{p} + \boldsymbol{I}(n)) : \boldsymbol{p} \in P(l)\}$$

Note that, ϕ_{isplit} ensures that statements in l_1 and l_2 correspond to those at locations in $P'(l_1)$ and $P'(l_2)$ respectively.

4 Verification

The loop transformation operates in three stages. First, the program text is turned into a polyhedral model. Then the model is transformed according to the specified script. Finally, the resulting model is turned back into a program text. The first (and more important in practice) check is to ensure that an incorrect script specified by a programmer is never executed. We do so by verifying certain conditions at the polyhedral level. The second is to ensure consistency between the execution semantics of the polyhedral model and its associated program. In our experience with Loopy, the first check has proved to be invaluable, preventing us from performing transformations that seemed correct but violated dependencies in subtle ways. The second check is not yet implemented.

4.1 Verifying the polyhedral transformation

Programmers have full freedom to specify transformations and may, therefore, specify incorrect ones. An incorrect specification will lead to a schedule that suffers from one of two possible problems:

- The schedule may not be a one-to-one map. I.e., more than one statement instance could be mapped to the same time in the new schedule.
- The new schedule may not respect execution order dependencies between statement instances. A transformation is guaranteed to be correct if it rearranges execution order only for independent statement instances. (Two statement instances are independent if they refer to disjoint variables, or if all common references are reads.)

The verifier in Loopy checks the correctness of a transformation by checking that it is a one-to-one map, and that it preserves dependencies. We assume that the original schedule is one-to-one and Loopy only checks one-to-oneness of the overall transformation function ϕ. Since ϕ is a linear transformation, Loopy uses a linear algebra library, ISL [26] to do this check. Loopy relies on the polyhedral model to supply the original execution order dependency maps between statement instances. If there is a dependency $s[\boldsymbol{i}] \rightarrow t[\boldsymbol{j}]$, then $s[\boldsymbol{i}]$ must be executed before $t[\boldsymbol{j}]$; a transformation that does not preserve this dependency is likely to be faulty. Let the final schedule of the transformed program be given by the function θ'. Loopy checks that $\theta'(s, \boldsymbol{i}) \prec \theta'(t, \boldsymbol{j})$ holds for all dependencies $s[\boldsymbol{i}] \rightarrow t[\boldsymbol{j}]$. If this check fails, the violated dependencies are reported to the user as a source of potential incorrectness.

The checks are performed only after composing the sequence of transformations in the specification. This is because dependencies can be temporarily violated in the middle of a sequence but established at the end. A check performed after every transformation would (incorrectly) mark such sequences as invalid. The situation is similar to that commonly encountered in establishing an inductive loop invariant, which may be temporarily violated within the loop body while being re-established at the start of the next iteration.

4.2 Verifying the program-model correspondence

We formulate the consistency question as follows. A program has a natural operational semantics, where the program is represented by a state transition system. The state is a map from variables to values, while a transition corresponds to execution of a statement instance. In the polyhedral model, on the other hand, statement instances are executed according to the specified schedule. The question is to formulate conditions under which the scheduled ordering coincides with the natural ordering. As will be apparent, the conditions have a strong similarity to invariants and ranking functions. This analysis is valid only for sequential programs, checking parallelizing transformations is a topic for future work.

As formulated in the previous section, polyhedral execution is defined over an iteration space I (the set of all valid iteration vectors). Each statement is associated over a subset of that space, called its domain (denoted $\mathsf{dom}(s)$, for statement s). The schedule function, $\theta : \mathit{stmt} \times I \rightarrow T$, maps a statement and a point of the space to an element of a totally ordered set, T ("time"). The schedule function is partial; however, for a statement s it is defined for all points in $\mathsf{dom}(s)$. The operational model executes statement instances in the order defined by the schedule. I.e., in a computation following the schedule, instance $t[\boldsymbol{j}]$ occurs after instance $s[\boldsymbol{i}]$ if $\theta(s, \boldsymbol{i}) \prec \theta(t, \boldsymbol{j})$.

As programs are sequential, one must rule out the possibility of concurrent execution. This is done by checking that the schedule is a one-to-one map, so that different instances are not mapped to the same time.. I.e., the following is valid:

$$[(s \neq t) \wedge (\boldsymbol{i} \neq \boldsymbol{j}) \wedge \boldsymbol{i} \in \mathsf{dom}(s) \wedge \boldsymbol{j} \in \mathsf{dom}(t) \Rightarrow \theta(s, \boldsymbol{i}) \neq \theta(t, \boldsymbol{j})]$$

Next, we present conditions which ensure that every natural execution is a scheduled execution. To simplify the presentation, we suppose that there are empty statements, entry and exit, at the start and end of the loop, with scheduled time $\bot$ (the minimum of the time domain) and $\top$ (the maximum of the time domain) respectively. Let $p_{s,t}$ represent the path transition relation from the state before statement s to the state before statement t.

(Inv) If an instance of t follows an instance of s in the program semantics, its iteration vector should belong to the domain of t. The following validity expresses the constraint.

$$[p_{s,t}(\boldsymbol{i},\boldsymbol{j}) \;\wedge\; \boldsymbol{i} \in \mathsf{dom}(s) \;\Rightarrow\; \boldsymbol{j} \in \mathsf{dom}(t)]$$

Note that this implies that the collection of statement domains is a mutual inductive invariant.

(Rank1) The instance of t must have a scheduled time after that of the instance of s. I.e., the following should be valid:

$$[p_{s,t}(\boldsymbol{i},\boldsymbol{j}) \;\wedge\; \boldsymbol{i} \in \mathsf{dom}(s) \;\Rightarrow\; \theta(s,\boldsymbol{i}) \prec \theta(t,\boldsymbol{j})]$$

(Rank2) The instance of t must have the *minimum* scheduled time after that of the instance of s. I.e., the following should be valid:

$$\begin{aligned}[p_{s,t}(\boldsymbol{i},\boldsymbol{j}) \;\wedge\; \boldsymbol{i} \in \mathsf{dom}(s) \;\wedge\; u \neq t \;\wedge\; \boldsymbol{k} \in \mathsf{dom}(u) \;\wedge\; \theta(s,\boldsymbol{i}) \prec \theta(u,\boldsymbol{k}) \\ \Rightarrow\; \theta(t,\boldsymbol{j}) \prec \theta(u,\boldsymbol{k})]\end{aligned}$$

These are implicitly universally quantified expressions in the free variables. As the domain and schedule functions are given by affine expressions, if the path conditions can be represented in an SMT-supported logic, the validity checks can be carried out automatically using an SMT solver.

To illustrate this further, consider the program in Figure 1a. As noted in the previous section, it has two statements, $s_{Init}[i,j]$ and $s_{Mult}[i,j,k]$. The iteration domains of these statements are $0 \leq i < N \wedge 0 \leq j < N$ and $0 \leq i < N \wedge 0 \leq j < N \wedge 0 \leq k < N$ and their initial schedule θ is $(0,i,0,j,0,0,0)$ and $(0,i,0,j,1,k,0)$ respectively. Now consider a path from $s_{Init}[i,j]$ to $s_{Mult}[i,j,0]$:

- Inv: For each $(i,j) \in \mathsf{dom}(s_{Init})$, $(i,j,0) \in \mathsf{dom}(s_{Mult})$
- Rank1: $\theta(s_{Init},(i,j)) = (0,i,0,j,0,0,0) \prec (0,i,0,j,1,0,0) = \theta(s_{Mult},(i,j,0))$
- Rank2: $s_{Mult}[i,j,0]$ has the minimum scheduled time after $s_{Init}[i,j]$

We similarly check paths entry to $s_{Init}[0,0]$, $s_{Mult}[i,j,k]$ to $s_{Mult}[i,j,k+1], k < N-1$, $s_{Mult}[i,j,N-1]$ to $s_{Init}[i,j+1], j < N-1$, and $s_{Mult}[N,N,N]$ to exit.

Theorem 1. *For a schedule meeting the conditions above, the scheduled computations are precisely the program computations.*

For any natural program computation, conditions (Inv), (Rank1) and (Rank2) ensure that there is a corresponding scheduled computation with the same sequence of actions. The other direction follows by determinism, non-blocking, and the 1-1 nature of θ. A detailed proof is presented in the appendix.

5 Implementation

We have implemented Loopy in Polly [10], a polyhedral library for LLVM [17]. LLVM is a popular compiler framework used to generate optimized code for various front-end high level languages and back-end platforms. LLVM converts the high-level program into an intermediate representation, known as the LLVM IR, that goes through a sequence of compiler analysis and transformations and is then converted into an executable for a back-end platform. Polly is a sophisticated library of transformations used to extract polyhedral models from LLVM IR, transform the extracted model, and convert it back to LLVM IR. Loopy is implemented as an optimization phase within Polly that transforms the polyhedral model of the program provided by Polly. Polyhedral models can be extracted for a wide variety of programs containing structured and unstructured loops [4]. Code is generated from the polyhedral model for various kinds of schedules [11]. By basing itself on Polly, Loopy becomes immediately applicable to a large class of programs and transformations.

ISL [26] is a library for representing and manipulating integer sets and relations; it supports various operations and decision procedures on these. This library is used to represent the components of polyhedral model, such as the set of statements, their iteration domains, and schedules. We further use ISL to implement the basic transformations, apply them on the polyhedral model, and to verify the final model for correctness. Verification checks the polyhedral transformation; the program-model correspondence is not yet implemented. Most of the operations used in defining our transformations can be mapped directly to an operation in ISL, which simplifies the implementation.

The optimization script is stored in a separate file, which is read in through a parser that extracts the type and inputs for each transformation to be applied on the program. Further, we rely on the ISL parser to parse the affine transformation and predicate representations for the *affine* and *isplit* operators.

To implement loop tags and extract domains of loop components in the polyhedral model, we do the following. The front-ends for LLVM do not necessarily preserve scope blocks while generating LLVM IR from the source code. Therefore, we surround a loop component block with a dummy `for` loop containing a single iteration, if there is not one already present. This helps identify the block of tagged code in the corresponding LLVM IR. While we currently perform this manually as needed, this will be automated in the future. The label of the first statement in a loop block is read during the construction of polyhedral model in Polly to get the desired loop component handle. We augment the polyhedral construction phase to construct a map from the handle to the initial schedule of all statements contained within the loop component, which gives the required domain of the loop component. This works well in general. However, we did find a few cases where prior optimization to LLVM IR either modified the label of the loop component or the loop itself and thus, domains could not be constructed for some of the loop components. We plan to resolve this issue in future.

6 Evaluation

In this section, we present a preliminary evaluation of Loopy. We had multiple goals while evaluating Loopy. First, we wanted to understand the amount of speed-up that can be achieved using Loopy as compared to state-of-the-art optimizing compilers. Second, we wanted to assess the amount of effort that is required to achieve these significant speed-ups. Third, we wanted to understand what kind of optimizations are applicable and which basic transformations get used most often. Lastly, we wanted to understand the overall experience of using this tool. We present our observations here.

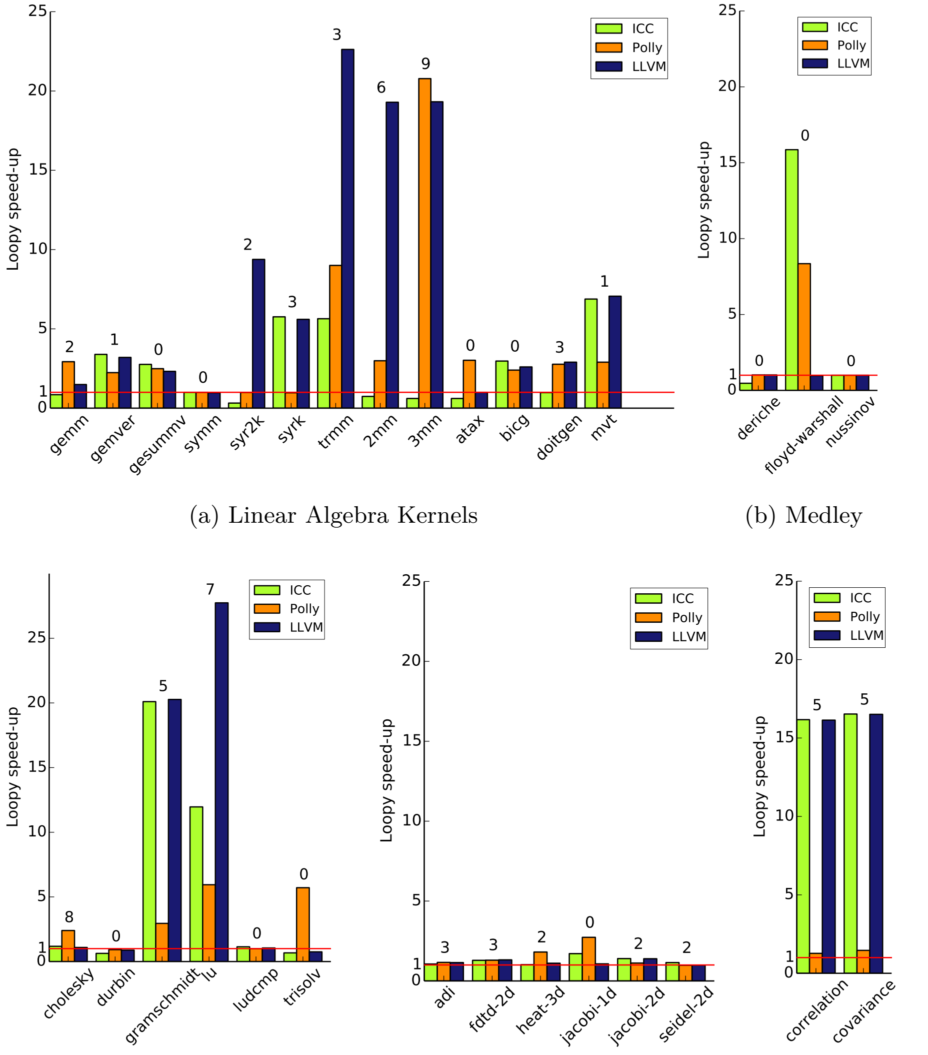
We evaluated Loopy on Polybench 4.1 [19], a benchmark suite maintained by the polyhedral compilation community. This is a collection of 30 programs from various domains, which provides a diverse set of benchmark programs. The programs expose only the kernels that need to be optimized and thus, are easy to work with. We optimize these programs for performance on single cores, focusing on improving cache usage. We compare Loopy with the PLUTO [5] based optimizer in Polly (LLVM version 3.7.0), with LLVM/Clang (version 3.7.0), and with the Intel C++ Compiler (version 16.0.2) under -O3 optimization.

For each program in the suite, we labeled the program with loop tags, and wrote an optimization script. We experimented with different combinations of transformations to improve performance of the programs, selecting those which showed significant improvements. All scripts together were written and analyzed in a *week's time*. They required combining specifications of loop splitting, loop merging, loop interchange, loop shifting and loop tiling. The specifications are included in the appendix. Most of the programs have simple optimization scripts.

The experimental setup used is as follows. We ran experiments on a Macbook Pro with Intel i5 processor (2.6 GHz, 3MB L3 Cache, 256KB L2 Cache) and 8 GB 1600 MHz DDR3 RAM. Polybench comes with data sets of different sizes, we use the standard data set here. We report execution times that are average of 3 runs of the programs, though we found them to be fairly consistent. The verification step does not noticeably influence the compilation time, therefore, we do not report the compilation times here.

The results of the experiments are shown in Figure 5. The benchmarks are split into different categories, with a sub-plot for each category. Each sub-plot shows the speed up achieved using Loopy-specified optimizations as compared to those performed by ICC, Polly and LLVM/clang, respectively. The number of basic transformations used by Loopy is annotated on top of the bars, for each benchmark.

As can be seen from the figure, we achieve significant speedups for Linear Algebra Kernels and Solvers and Data Mining applications. We found that most of these programs were variants of matrix multiplication program from Figure 1a, and similar ideas worked to improve the performance. In particular, improving spatial locality of data accesses resulted in a significant speed up. Tiling the iteration space was also useful in improving the temporal locality, although it improved the performance only slightly. In this class, the optimization scripts were fairly small and involved realigning loops followed by affine transformations.



(a) Linear Algebra Kernels

(b) Medley

(c) Linear Algebra Solvers

(d) Stencils

(e) Data Mining

Fig. 5: Speed-ups with Loopy vs ICC, Polly, and LLVM/clang on Polybench programs. The number of basic transformations used by Loopy is annotated on top of the bars, for each benchmark. Speed-ups are ratios of execution times of the kernels: e.g., speedup in Loopy vs ICC is given by (ICC optimized execution time)/(Loopy optimized execution time).

We did not achieve significant speedups for Stencils, as the data-accesses were already aligned well and hence exhibited good spatial locality. Most of the speed up came from shifting loops and then merging adjacent loops that led to reuse of data elements across loops and hence improved temporal locality. In some cases, it is possible to tile the loops; however, this required complex skewing transformations, which we found hard to specify manually. Specifically, in `seidel-2d`, it took multiple iterations to find the right transformation and verification came handy in this search. We plan to experiment with new operators which address loop skewing directly, which should simplify this process in future.

Lastly, for some benchmarks, we did not find any transformation (in the limited time spent optimizing the scripts) that improved the performance of the application. In Figure 5, those are applications where the number of transformations is 0. Note that in all cases, our performance is almost equivalent or better than the other compilers. This indicates that those programs were difficult to optimize automatically as well. Further in certain cases, automatic optimization deteriorated performance. For example in linear algebra kernels `gesummv` and `bicg`, Polly tiles a key loop in the program which already exhibits good spatial and temporal locality. This transformation, therefore, only adds an overhead of additional checks and leads to poor performance.

We now detail some of experiences with using the tool. First, we found verification to be helpful in finding bugs in our optimizations. In particular, while optimizing `lu`, we received an error on merging adjacent loops, which we had thought initially was possible. On further inspection, we realized that Polly normalizes iterators to start from 0 and increment by 1, which caused the problem with our strategy. After modifying the transformation to take this into account, we obtained a correct optimization. The feedback about which dependencies were violated in the initial strategy was helpful in understanding the source of the mistake. For programs `3mm` and `ludcmp`, we found that Polly does not produce dependency maps between statements, which prevents Polly from optimizing the program. For `Loopy`, on the other hand, dependency maps are only necessary to check correctness at the final step, and are not needed for the actual transformation. We were able to transform those programs and achieved significant speedup for `3mm`. Lastly, for `durbin` and `ludcmp`, we found that some of the loop tags in the source program are not preserved in the conversion to LLVM IR. This prevented us from applying certain optimizations.

7 Related Work

Our system builds upon a considerable body of work on automated loop transformation methods, which is covered in a number of excellent books [2,1,18,3], papers and tools. In particular, we build upon the theory of polyhedral loop transformation (cf. [8,14]) – which grew out of earlier work on algebraic representations of iteration spaces [16] and other influences – and its implementation in the Polly system for LLVM [10]. A well studied means for automated optimization (e.g., followed in PLUTO [5]) is to convert the problem into an integer linear programming (ILP) formulation and optimize an objective function. A different approach (cf. [28,9,15,25,24]) is a search through the space of sequences of transformations: the search process generates sequences of transformations formed from a basis set and chooses those which maximize performance based on run-time tests. These automatic approaches work well in a variety of situations, though not *all* situations, primarily due to the computational hardness of the underlying problems. Loopy relies on programmers to specify transformations and programmer insight can often supersede these sophisticated algorithms.

A few domain specific works have also been developed along similar lines: SPIRAL [20] for digital signal processing, Halide [21] for image processing pipelines and BLAC [23] for linear algebra expressions. However, first, they are specific to a domain, and second, they require programmers to express computation in a new language which might be a steep learning curve.

Other systems have been developed with goals similar to ours. CHiLL [6,22] is a system which makes available a rich set of affine transformations to the programmer. POET [28], Orio [12] and X Language [7] also provide similar facilities. However, as noted in Section 2, these systems do not guard programmers against incorrect specifications, which may hinder usability. Another system from ALCHEMY group, URUK [9] provides transformation primitives that operate on polyhedral representation of loops and also checks transformations for correctness. However, polyhedral representation can be too abstract for a programmer to use efficiently. In fact, the primary motive of this work was to provide a structured transformation space for automatic methods to search efficient implementations, and not direct usage by programmers. Loopy represents the best of both worlds with clean programmer semantics and strong correctness guarantees.

8 Conclusion and Future Work

The Loopy framework gives full freedom to a programmer's ingenuity while ensuring that every transformation is correctly implemented. The key insight is that the combination of flexibility, automation and checking ("trust but verify"!) is powerful and enjoyable to work with. Our experiments show that simple, direct specifications can result in significant improvements over fully automated methods. In future work, we plan to explore this combination for the *parallelization* of loops for multi-core and GPU platforms. Here, verification becomes even more essential, in order to catch subtle errors that can arise from weak memory models. We hope to be able to make use of the considerable literature on weak memory model verification to ensure correct transformations.

Acknowledgements. We would like to thank our colleagues at Bell Labs and at the University of Pennsylvania for their helpful comments on this research. This work was supported, in part, by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2006)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
3. Banerjee, U.: Dependence analysis. Loop transformations for restructuring compilers, Kluwer (1997)
4. Benabderrahmane, M., Pouchet, L., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. pp. 283–303 (2010), http://dx.doi.org/10.1007/978-3-642-11970-5_16
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. pp. 101–113 (2008), http://doi.acm.org/10.1145/1375581.1375595
6. Chen, C., Chame, J., Hall, M.: CHiLL: A framework for composing high-level loop transformations. Tech. Rep. 08-897, University of Southern California (2008)
7. Donadio, S., Brodman, J., Roeder, T., Yotov, K., Barthou, D., Cohen, A., Garzarán, M.J., Padua, D., Pingali, K.: A language for the compact representation of multiple program versions. In: Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing. pp. 136–151. LCPC'05, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/978-3-540-69330-7_10
8. Feautrier, P.: Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. International Journal of Parallel Programming 21(6), 389–420 (1992), http://dx.doi.org/10.1007/BF01379404
9. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. International Journal of Parallel Programming 34(3), 261–317 (2006), http://dx.doi.org/10.1007/s10766-006-0012-3
10. Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters 22(4) (2012), http://dx.doi.org/10.1142/S0129626412500107
11. Grosser, T., Verdoolaege, S., Cohen, A.: Polyhedral AST generation is more than scanning polyhedra. ACM Trans. Program. Lang. Syst. 37(4), 12 (2015), http://doi.acm.org/10.1145/2743016
12. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using Orio. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. pp. 1–11 (2009), http://dx.doi.org/10.1109/IPDPS.2009.5161004
13. Intel: Intel Math Kernel Library (MKL) (2016), https://software.intel.com/en-us/intel-mkl/
14. Kelly, W., Pugh, W.: A framework for unifying reordering transformations. Tech. Rep. UMIAS-TR-92-126.1, Univ. of Maryland, College Park, MD, USA (1993)
15. Khan, M.M., Basu, P., Rudy, G., Hall, M.W., Chen, C., Chame, J.: A script-based autotuning compiler system to generate high-performance CUDA code. TACO 9(4), 31 (2013), http://doi.acm.org/10.1145/2400682.2400690

16. Lamport, L.: The parallel execution of DO loops. Commun. ACM 17(2), 83–93 (1974), http://doi.acm.org/10.1145/360827.360844
17. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75–88 (2004), webpage at llvm.org
18. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
19. Pouchet, L.N.: Polybench, the polyhedral benchmark suite (2015), http://polybench.sourceforge.net/
20. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93(2), 232– 275 (2005)
21. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 519–530. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462176
22. Rudy, G., Khan, M.M., Hall, M.W., Chen, C., Chame, J.: A programming language interface to describe transformations and code generation. In: Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers. pp. 136–150 (2010), http://dx.doi.org/10.1007/978-3-642-19595-2_10
23. Spampinato, D.G., Püschel, M.: A basic linear algebra compiler. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 23:23–23:32. CGO '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2544137.2544155
24. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. SIGPLAN Not. 50(9), 205–217 (Aug 2015), http://doi.acm.org/10.1145/2858949.2784754
25. Tiwari, A., Chen, C., Chame, J., Hall, M.W., Hollingsworth, J.K.: A scalable autotuning framework for compiler optimization. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. pp. 1–12 (2009), http://dx.doi.org/10.1109/IPDPS.2009.5161054
26. Verdoolaege, S.: *isl*: An integer set library for the polyhedral model. In: Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings. pp. 299–302 (2010), http://dx.doi.org/10.1007/978-3-642-15582-6_49
27. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing. pp. 1–27. SC '98, IEEE Computer Society, Washington, DC, USA (1998), http://dl.acm.org/citation.cfm?id=509058.509096
28. Yi, Q.: POET: a scripting language for applying parameterized source-to-source program transformations. Softw., Pract. Exper. 42(6), 675–706 (2012), http://dx.doi.org/10.1002/spe.1089

A Proofs

The computation graph induced by a schedule function θ from a state s_0 is defined as follows. Graph nodes are called *configurations*, each is a pair: (state, time). The initial configuration is $(s_0, 0)$. A configuration (s', T') is a successor of configuration (s, T) by an instance (a, k) if (1) (a, k) has the least scheduled time beyond T, (2) $s' = a(s[I \leftarrow k])$. I.e., s' is obtained by applying action a to a modification of state s where the iteration variables I are given values by vector k, and (3) $T' = \theta(a, k)$.

Proof of Theorem 1. The proof shows that the natural and scheduled computations are identical from a common initial state, s_0. There is a single natural computation, η, from s_0, as the program is deterministic. There is also a single scheduled computation, σ, from s_0, as the schedule function is 1-1, so there is a unique next action at each step. Let $\eta = s_0; a_0; s_1; a_1; \ldots$, where the s_i's are states and the a_i's are actions. For any k, let $\eta[k]$ denote the prefix of η with the first k actions, and similarly for $\sigma[k]$. (For values of k that are greater than the number of actions on a (finite) computation, the prefix is the computation itself.) Let the iteration vector at state x be denoted by $\mathsf{iv}(x)$.

The proof is by induction. The inductive claim is that for every k, the prefixes $\eta[k]$ and $\sigma[k]$ are identical, and that for each i (with $0 \leq i < k$), if a_i is defined, then $\mathsf{iv}(s_i)$ is in $\mathsf{dom}(a_i)$, and the action in σ corresponding to a_i is $(a_i, \mathsf{iv}(s_i))$.

This claim is true trivially for $k = 0$. It is also true trivially for $k = 1$, as the first action must be the entry action, which has the smallest scheduled time and whose domain contains only the initial iteration vector at s_0. Let $k > 0$ and assume that the inductive claim holds for all values up to k. Consider the claim for $k + 1$. There are three cases:

(a) The natural computation has l actions, where $l < k$. Then $\eta[k+1]$, $\eta[k]$ and $\eta[l]$ are identical. From the inductive assumption, $\eta[l] = \sigma[l]$ and $\eta[k] = \sigma[k]$, so that $\sigma[l] = \sigma[k]$. Hence, the scheduled computation has l actions as well, which implies that $\sigma[k+1] = \sigma[k]$. Therefore, $\sigma[k+1] = \eta[k+1]$ and the claim is established for $k + 1$.

(b) The natural computation has k actions, so that $\eta[k+1] = \eta[k]$. Then s_k is the final state on η, and the last action on $\eta[k]$ and $\sigma[k]$ – which are identical by the induction hypothesis – must be the exit action. As this action has the maximum scheduled time, there is no successor to s_k on the scheduled computation, either. Thus, $\sigma[k+1] = \sigma[k]$; hence, $\sigma[k+1] = \eta[k+1]$ and the claim is established for $k + 1$.

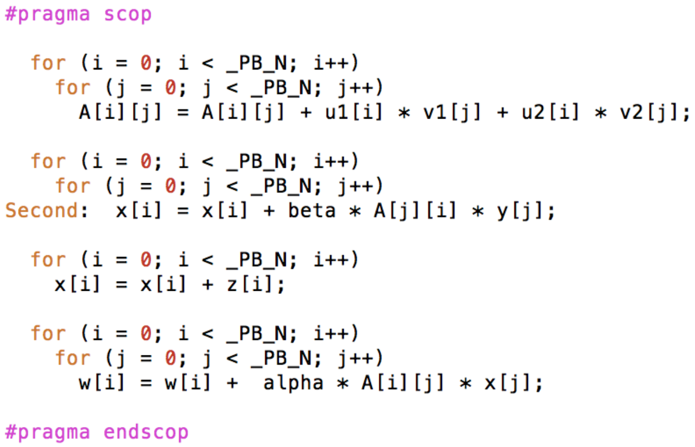
(c) The natural computation has more than k actions, so that s_k is not the final state. Let $a = a_{k-1}$ and $b = a_k$. The path condition $p_{a,b}$ holds for $(\mathsf{iv}(s_{k-1}), \mathsf{iv}(s_k))$. By the inductive assumption, $\mathsf{iv}(s_{k-1})$ is in $\mathsf{dom}(a_{k-1})$ so, by (Inv), $\mathsf{iv}(s_k)$ is in $\mathsf{dom}(a_k)$. By (Rank1) and (Rank2), $(a_k, \mathsf{iv}(s_k))$ is the action instance with the least scheduled time after $\theta(a_{k-1}, \mathsf{iv}(s_{k-1}))$. Hence, $(a_k, \mathsf{iv}(s_k))$ is the k'th next action on σ, with successor state s_{k+1}. This re-establishes the inductive claim for $k + 1$. **EndProof.**

B Optimization Scripts

We present here some optimization scripts used to optimize Polybench programs. We also present the annotated kernels corresponding to the optimization scripts. The benchmarks presented here are: gemver, trmm, 2mm, doitgen, 3mm, mvt, gramschmidt, lu, adi, fdtd-2d, heat-3d, jacobi-2d, seidel-2d and covariance.

1. **linear-algebra / blas / gemver**:

   Annotated Kernel:



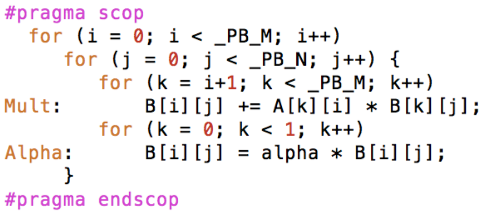
```
#pragma scop

  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      A[i][j] = A[i][j] + u1[i] * v1[j] + u2[i] * v2[j];

  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
Second:  x[i] = x[i] + beta * A[j][i] * y[j];

  for (i = 0; i < _PB_N; i++)
    x[i] = x[i] + z[i];

  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
      w[i] = w[i] +  alpha * A[i][j] * x[j];

#pragma endscop
```

   Optimization:

```
affine(Second, {[i,j]->[j,i]})
```

2. **linear-algebra / blas / trmm**:

   Annotated Kernel:



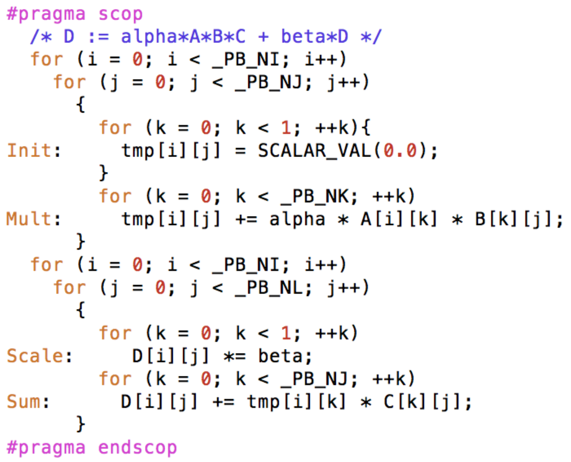
```
#pragma scop
  for (i = 0; i < _PB_M; i++)
     for (j = 0; j < _PB_N; j++) {
        for (k = i+1; k < _PB_M; k++)
Mult:       B[i][j] += A[k][i] * B[k][j];
        for (k = 0; k < 1; k++)
Alpha:      B[i][j] = alpha * B[i][j];
     }
#pragma endscop
```

   Optimization:

```
realign(Mult, Alpha, 0)
affine(Mult, {[i,j,k]->[i,k,j]})
affine(Mult, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32 and k1=[k/32] and k2=k%32})
```

3. **linear-algebra / kernels / 2mm**:

   Annotated Kernel:

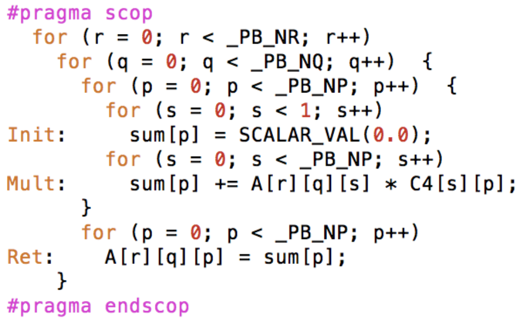


```
#pragma scop
  /* D := alpha*A*B*C + beta*D */
  for (i = 0; i < _PB_NI; i++)
    for (j = 0; j < _PB_NJ; j++)
      {
        for (k = 0; k < 1; ++k){
Init:     tmp[i][j] = SCALAR_VAL(0.0);
        }
        for (k = 0; k < _PB_NK; ++k)
Mult:     tmp[i][j] += alpha * A[i][k] * B[k][j];
      }
  for (i = 0; i < _PB_NI; i++)
    for (j = 0; j < _PB_NL; j++)
      {
        for (k = 0; k < 1; ++k)
Scale:     D[i][j] *= beta;
        for (k = 0; k < _PB_NJ; ++k)
Sum:      D[i][j] += tmp[i][k] * C[k][j];
      }
#pragma endscop
```

Optimization:

```
realign(Init, Mult, 0)
affine(Mult, {[i,j,k]->[i,k,j]})
affine(Mult, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32 and k1=[k/32] and k2=k%32})

realign(Scale, Sum, 0)
affine(Sum, {[i,j,k]->[i,k,j]})
affine(Sum, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32 and k1=[k/32] and k2=k%32})
```

4. **linear-algebra / kernels / doitgen**:
   Annotated Kernel:

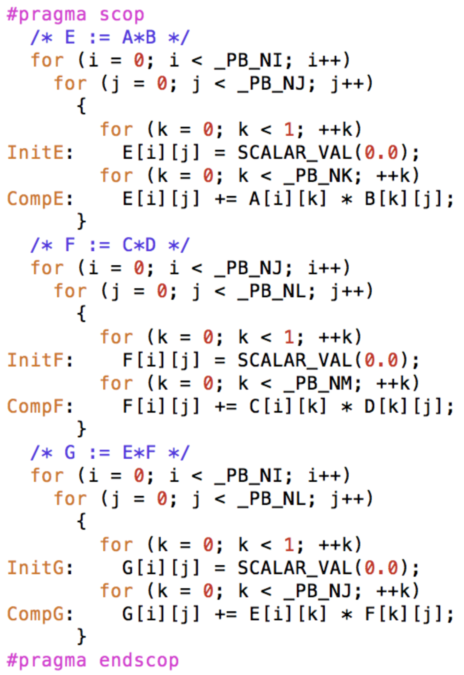


```
#pragma scop
  for (r = 0; r < _PB_NR; r++)
    for (q = 0; q < _PB_NQ; q++)  {
      for (p = 0; p < _PB_NP; p++)  {
        for (s = 0; s < 1; s++)
Init:     sum[p] = SCALAR_VAL(0.0);
        for (s = 0; s < _PB_NP; s++)
Mult:     sum[p] += A[r][q][s] * C4[s][p];
      }
      for (p = 0; p < _PB_NP; p++)
Ret:    A[r][q][p] = sum[p];
    }
#pragma endscop
```

Optimization:

```
realign(Init, Mult, 2)
affine(Mult, {[r,q,p,s]->[r,q,s,p]})
affine(Mult, {[r,q,s,p]->[r,q,s1,p1,s2,p2]: p1=[p/64] and p2=p%64 and s1=[s/64]
and s2=s%64})
```

5. **linear-algebra / kernels / 3mm**:
   Annotated Kernel:



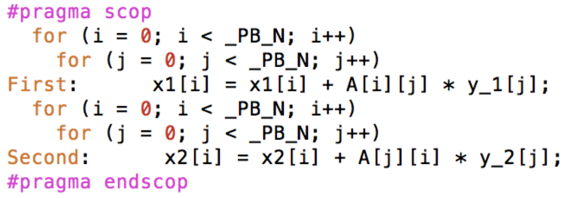
```
#pragma scop
  /* E := A*B */
  for (i = 0; i < _PB_NI; i++)
    for (j = 0; j < _PB_NJ; j++)
      {
        for (k = 0; k < 1; ++k)
InitE:    E[i][j] = SCALAR_VAL(0.0);
        for (k = 0; k < _PB_NK; ++k)
CompE:    E[i][j] += A[i][k] * B[k][j];
      }
  /* F := C*D */
  for (i = 0; i < _PB_NJ; i++)
    for (j = 0; j < _PB_NL; j++)
      {
        for (k = 0; k < 1; ++k)
InitF:    F[i][j] = SCALAR_VAL(0.0);
        for (k = 0; k < _PB_NM; ++k)
CompF:    F[i][j] += C[i][k] * D[k][j];
      }
  /* G := E*F */
  for (i = 0; i < _PB_NI; i++)
    for (j = 0; j < _PB_NL; j++)
      {
        for (k = 0; k < 1; ++k)
InitG:    G[i][j] = SCALAR_VAL(0.0);
        for (k = 0; k < _PB_NJ; ++k)
CompG:    G[i][j] += E[i][k] * F[k][j];
      }
#pragma endscop
```

Optimization:

```
realign(InitE, CompE, 0)
affine(CompE, {[i,j,k]->[i,k,j]})
affine(CompE, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and
j1=[j/32] and j2=j%32 and k1=[k/32] and k2=k%32})

realign(InitF, CompF, 0)
affine(CompF, {[i,j,k]->[i,k,j]})
affine(CompF, {[i,j,k]->[i1, j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and
j1=[j/32] and j2=j%32 and k1=[k/32] and k2=k%32})

realign(InitG, CompG, 0)
affine(CompG, {[i,j,k]->[i,k,j]})
affine(CompG, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and
j1=[j/32] and j2=j%32 and k1=[k/32] and k2=k%32})
```

6. **linear-algebra / kernels / mvt**:

Annotated Kernel:



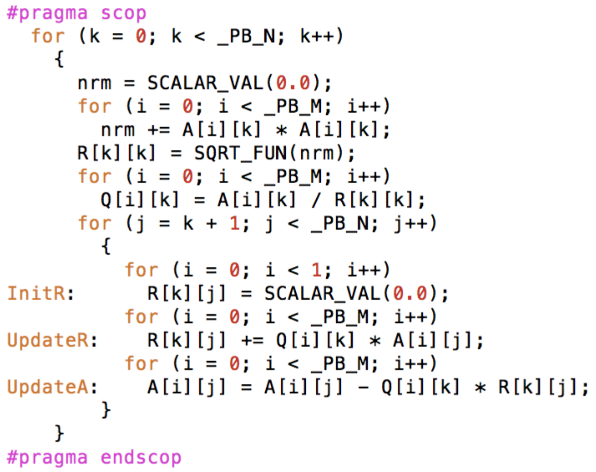
```
#pragma scop
  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
First:      x1[i] = x1[i] + A[i][j] * y_1[j];
  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_N; j++)
Second:      x2[i] = x2[i] + A[j][i] * y_2[j];
#pragma endscop
```

Optimization:

```
affine(Second, {[i,j]->[j,i]})
```

7. **linear-algebra / solvers / gramschmidt**:

   Annotated Kernel:



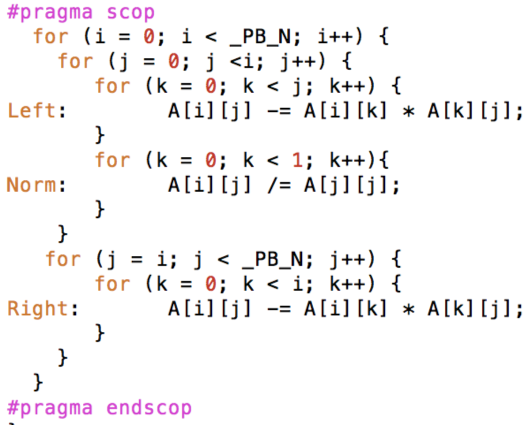
```
#pragma scop
  for (k = 0; k < _PB_N; k++)
    {
      nrm = SCALAR_VAL(0.0);
      for (i = 0; i < _PB_M; i++)
        nrm += A[i][k] * A[i][k];
      R[k][k] = SQRT_FUN(nrm);
      for (i = 0; i < _PB_M; i++)
        Q[i][k] = A[i][k] / R[k][k];
      for (j = k + 1; j < _PB_N; j++)
        {
          for (i = 0; i < 1; i++)
InitR:      R[k][j] = SCALAR_VAL(0.0);
          for (i = 0; i < _PB_M; i++)
UpdateR:    R[k][j] += Q[i][k] * A[i][j];
          for (i = 0; i < _PB_M; i++)
UpdateA:    A[i][j] = A[i][j] - Q[i][k] * R[k][j];
        }
    }
#pragma endscop
```

   Optimization:

```
realign(InitR, UpdateR, 1)
realign(UpdateR, UpdateA, 1)
affine(InitR, {[i,j,k]->[i,k,j]})
affine(UpdateR, {[i,j,k]->[i,k,j]})
affine(UpdateA, {[i,j,k]->[i,k,j]})
```

8. **linear-algebra / solvers / lu**:

   Annotated Kernel:



```
#pragma scop
  for (i = 0; i < _PB_N; i++) {
    for (j = 0; j <i; j++) {
       for (k = 0; k < j; k++) {
Left:        A[i][j] -= A[i][k] * A[k][j];
       }
       for (k = 0; k < 1; k++){
Norm:        A[i][j] /= A[j][j];
       }
    }
   for (j = i; j < _PB_N; j++) {
       for (k = 0; k < i; k++) {
Right:       A[i][j] -= A[i][k] * A[k][j];
       }
    }
  }
#pragma endscop
```

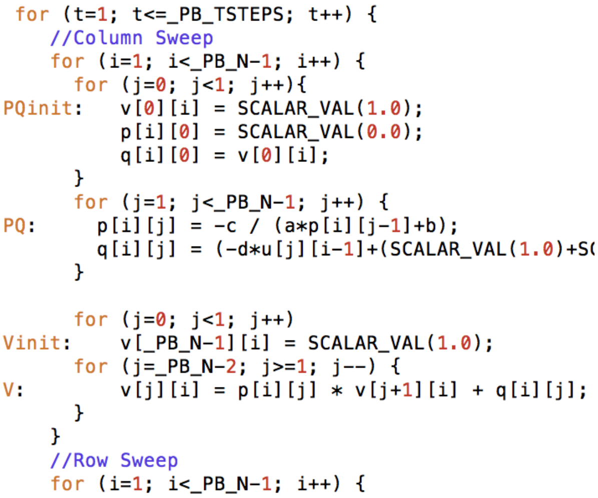
   Optimization:

```
affine(Right, {[i,j,k]->[i,j+i,k]})
affine(Norm, {[i,j,0]->[i,j,j]})
```

```
realign(Norm, Right, 3)
realign(Left, Norm, 3)
L = lift(Left, 3)
affine(L, {[i,j,k]->[i,k,j]})
affine(L, {[i,j,k]->[i1,j1,k1,i2,j2,k2]:
i1=[i/128] and i2=i%128 and j1=[j/128] and j2=j%128 and k1=[k/128] and
k2=k%128})
```

9. **stencils / adi**:

Annotated Kernel:



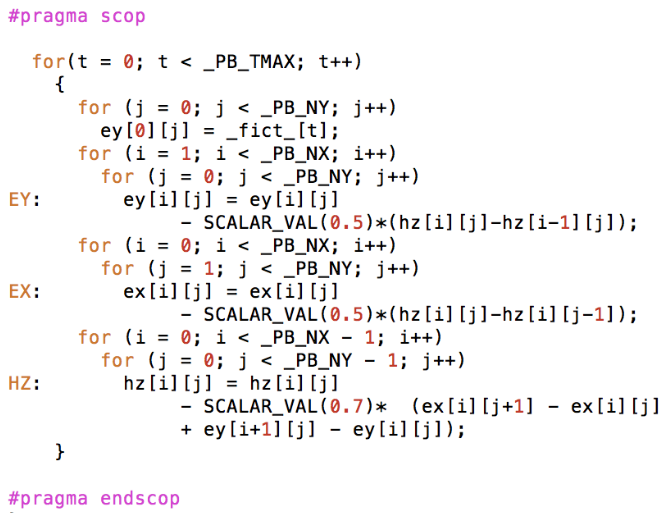
```
 for (t=1; t<=_PB_TSTEPS; t++) {
    //Column Sweep
    for (i=1; i<_PB_N-1; i++) {
      for (j=0; j<1; j++){
PQinit:   v[0][i] = SCALAR_VAL(1.0);
          p[i][0] = SCALAR_VAL(0.0);
          q[i][0] = v[0][i];
      }
      for (j=1; j<_PB_N-1; j++) {
PQ:     p[i][j] = -c / (a*p[i][j-1]+b);
        q[i][j] = (-d*u[j][i-1]+(SCALAR_VAL(1.0)+S(
      }

      for (j=0; j<1; j++)
Vinit:    v[_PB_N-1][i] = SCALAR_VAL(1.0);
      for (j=_PB_N-2; j>=1; j--) {
V:        v[j][i] = p[i][j] * v[j+1][i] + q[i][j];
      }
    }
    //Row Sweep
    for (i=1; i<_PB_N-1; i++) {
```

Optimization:

```
realign(Vinit, V, 1)
affine(V, {[i,j,k]->[i,k,j]})
affine(V, {[t,i,j]->[t,i1,j1,i2,j2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32})
```

10. **stencils / fdtd-2d**:

Annotated Kernel:



```
#pragma scop

  for(t = 0; t < _PB_TMAX; t++)
    {
      for (j = 0; j < _PB_NY; j++)
        ey[0][j] = _fict_[t];
      for (i = 1; i < _PB_NX; i++)
        for (j = 0; j < _PB_NY; j++)
EY:       ey[i][j] = ey[i][j]
               - SCALAR_VAL(0.5)*(hz[i][j]-hz[i-1][j]);
      for (i = 0; i < _PB_NX; i++)
        for (j = 1; j < _PB_NY; j++)
EX:       ex[i][j] = ex[i][j]
               - SCALAR_VAL(0.5)*(hz[i][j]-hz[i][j-1]);
      for (i = 0; i < _PB_NX - 1; i++)
        for (j = 0; j < _PB_NY - 1; j++)
HZ:       hz[i][j] = hz[i][j]
               - SCALAR_VAL(0.7)*  (ex[i][j+1] - ex[i][j]
               + ey[i+1][j] - ey[i][j]);
    }

#pragma endscop
```

Optimization:

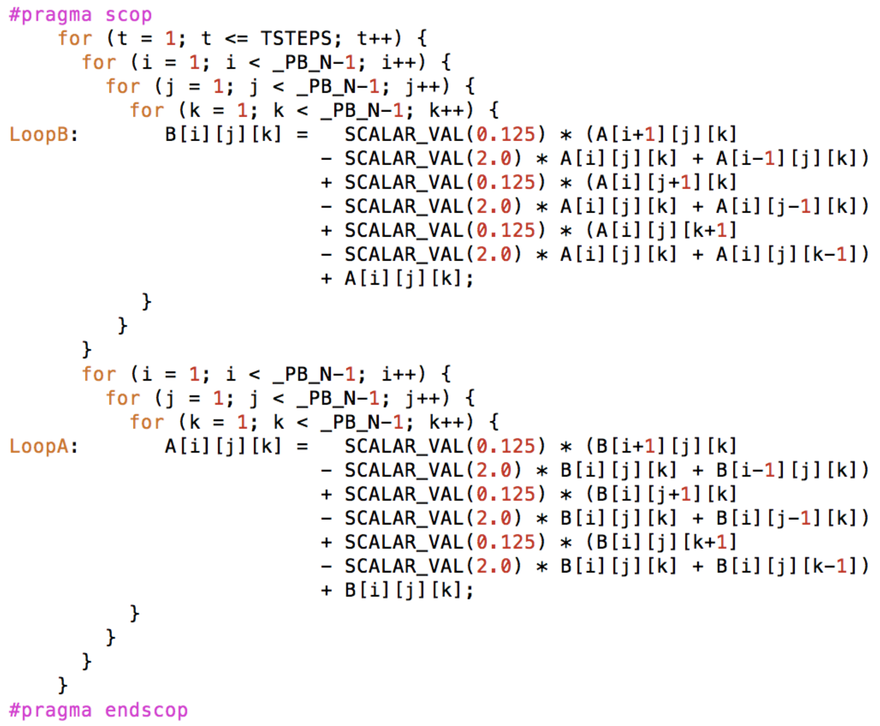
`realign(EY, EX, 3)`

`affine(HZ, {[t,i,j]->[t,i+1,j]})`

`realign(EX, HZ, 3)`

11. **stencils / heat-3d**:

Annotated Kernel:



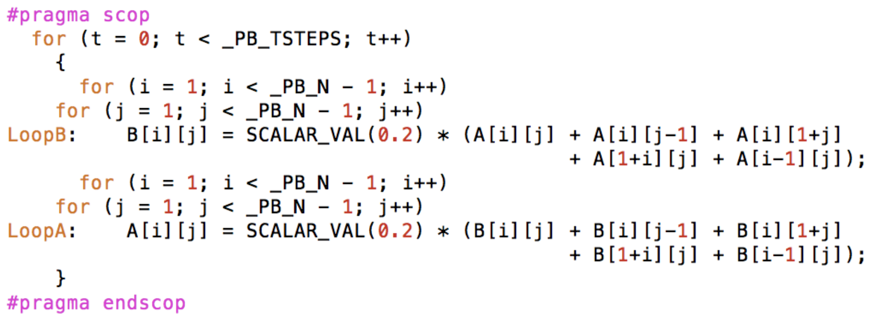
```
#pragma scop
    for (t = 1; t <= TSTEPS; t++) {
      for (i = 1; i < _PB_N-1; i++) {
        for (j = 1; j < _PB_N-1; j++) {
          for (k = 1; k < _PB_N-1; k++) {
LoopB:       B[i][j][k] =   SCALAR_VAL(0.125) * (A[i+1][j][k]
                          - SCALAR_VAL(2.0) * A[i][j][k] + A[i-1][j][k])
                          + SCALAR_VAL(0.125) * (A[i][j+1][k]
                          - SCALAR_VAL(2.0) * A[i][j][k] + A[i][j-1][k])
                          + SCALAR_VAL(0.125) * (A[i][j][k+1]
                          - SCALAR_VAL(2.0) * A[i][j][k] + A[i][j][k-1])
                          + A[i][j][k];
           }
         }
      }
      for (i = 1; i < _PB_N-1; i++) {
        for (j = 1; j < _PB_N-1; j++) {
          for (k = 1; k < _PB_N-1; k++) {
LoopA:       A[i][j][k] =   SCALAR_VAL(0.125) * (B[i+1][j][k]
                          - SCALAR_VAL(2.0) * B[i][j][k] + B[i-1][j][k])
                          + SCALAR_VAL(0.125) * (B[i][j+1][k]
                          - SCALAR_VAL(2.0) * B[i][j][k] + B[i][j-1][k])
                          + SCALAR_VAL(0.125) * (B[i][j][k+1]
                          - SCALAR_VAL(2.0) * B[i][j][k] + B[i][j][k-1])
                          + B[i][j][k];
          }
        }
      }
    }
#pragma endscop
```

Optimization:

```
affine(LoopA, {[t,i,j,k]->[t,i+1,j,k]})
realign(LoopB, LoopA, 4)
```

12. **stencils / jacobi-2d**:

    Annotated Kernel:



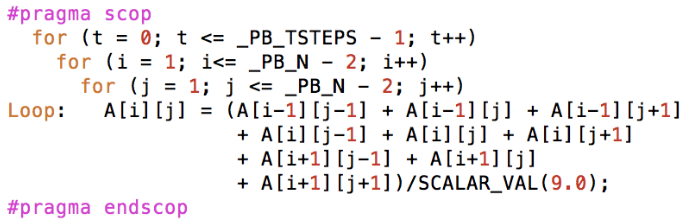
```
#pragma scop
  for (t = 0; t < _PB_TSTEPS; t++)
    {
      for (i = 1; i < _PB_N - 1; i++)
    for (j = 1; j < _PB_N - 1; j++)
LoopB:    B[i][j] = SCALAR_VAL(0.2) * (A[i][j] + A[i][j-1] + A[i][1+j]
                                               + A[1+i][j] + A[i-1][j]);
      for (i = 1; i < _PB_N - 1; i++)
    for (j = 1; j < _PB_N - 1; j++)
LoopA:    A[i][j] = SCALAR_VAL(0.2) * (B[i][j] + B[i][j-1] + B[i][1+j]
                                               + B[1+i][j] + B[i-1][j]);
    }
#pragma endscop
```

    Optimization:

```
affine(LoopA, {[t,i,j]->[t,i+1,j]})
realign(LoopB, LoopA, 3)
```

13. **stencils / seidel-2d**:

    Annotated Kernel:



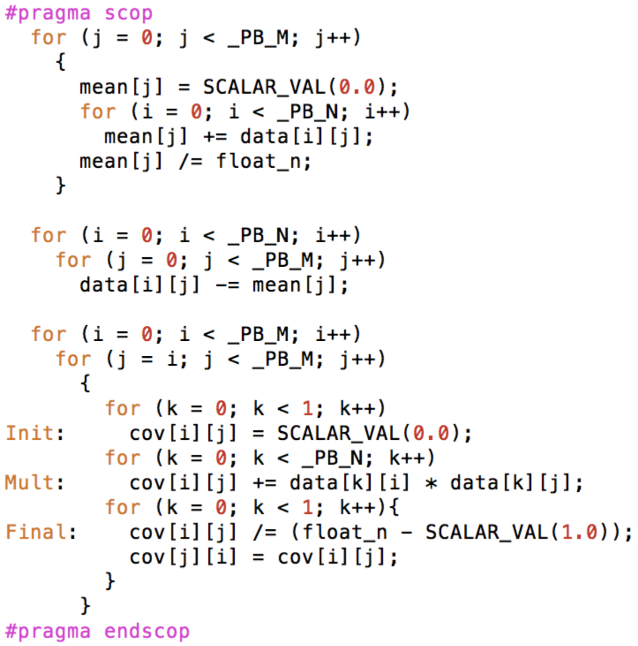
```
#pragma scop
  for (t = 0; t <= _PB_TSTEPS - 1; t++)
    for (i = 1; i<= _PB_N - 2; i++)
      for (j = 1; j <= _PB_N - 2; j++)
Loop:   A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
                   + A[i][j-1] + A[i][j] + A[i][j+1]
                   + A[i+1][j-1] + A[i+1][j]
                   + A[i+1][j+1])/SCALAR_VAL(9.0);
#pragma endscop
```

    Optimization:

```
affine(Loop, {[t,i,j]->[t,i+t,j+i]})
affine(Loop, {[t,i,j]->[t,i1,j1,i2,j2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32})
```

14. **data-mining / covariance**:

    Annotated Kernel:



```
#pragma scop
  for (j = 0; j < _PB_M; j++)
    {
      mean[j] = SCALAR_VAL(0.0);
      for (i = 0; i < _PB_N; i++)
        mean[j] += data[i][j];
      mean[j] /= float_n;
    }

  for (i = 0; i < _PB_N; i++)
    for (j = 0; j < _PB_M; j++)
      data[i][j] -= mean[j];

  for (i = 0; i < _PB_M; i++)
    for (j = i; j < _PB_M; j++)
      {
        for (k = 0; k < 1; k++)
Init:     cov[i][j] = SCALAR_VAL(0.0);
        for (k = 0; k < _PB_N; k++)
Mult:     cov[i][j] += data[k][i] * data[k][j];
        for (k = 0; k < 1; k++){
Final:    cov[i][j] /= (float_n - SCALAR_VAL(1.0));
          cov[j][i] = cov[i][j];
        }
      }
#pragma endscop
```

Optimization:

```
realign(Init, Mult, 0)
realign(Mult, Final, 0)
affine(Mult, {[i,j,k]->[k,i,j]})
affine(Mult, {[i,j,k]->[i1,j1,k1,i2,j2,k2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32 and k1=[k/32] and k2=k%32})
affine(Final, {[i,j]->[i1,j1,i2,j2]: i1=[i/32] and i2=i%32 and j1=[j/32]
and j2=j%32})
```