PolyCheck: Dynamic Verification of Iteration Space Transformations on Affine Programs

Wenlei Bao
The Ohio State University, USA
bao.79@osu.edu

Sriram Krishnamoorthy
Pacific Northwest National Laboratory, USA
sriram@pnnl.gov

Louis-Noël Pouchet
The Ohio State University, USA
pouchet@cse.ohio-state.edu

Fabrice Rastello
INRIA, France
Fabrice.Rastello@inria.fr

P. Sadayappan
The Ohio State University, USA
saday@cse.ohio-state.edu

Abstract

High-level compiler transformations, especially loop transformations, are widely recognized as critical optimizations to restructure programs to improve data locality and expose parallelism. Guaranteeing the correctness of program transformations is essential, and to date three main approaches have been developed: proof of equivalence of affine programs, matching the execution traces of programs, and checking bit-by-bit equivalence of program outputs. Each technique suffers from limitations in the kind of transformations supported, space complexity, or the sensitivity to the testing dataset. In this paper, we take a novel approach that addresses all three limitations to provide an automatic bug checker to verify any iteration reordering transformations on affine programs, including non-affine transformations, with space consumption proportional to the original program data and robust to arbitrary datasets of a given size. We achieve this by exploiting the structure of affine program control- and data-flow to generate at compile-time lightweight checker code to be executed within the transformed program. Experimental results assess the correctness and effectiveness of our method and its increased coverage over previous approaches.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, verification

Keywords Dynamic verification, iteration space transformation, static analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837656

1. Introduction

Optimized programs are often complex and rarely resemble the original source programs. It is difficult, if not impossible, to verify the correctness of such programs through visual inspection. Automated testing is part of the classical arsenal to find bugs in compiler implementations, including production compilers [2, 5] and research compilers [41]. However, a typical challenge occurs with the selection of a relevant dataset for the program to expose a bug. If we limit testing to the bit-by-bit equivalence of outputs produced by the original and transformed programs, what guarantees that a match implies the lack of a bug? A simplistic example would entail testing a program multiplying two matrices, where the input matrices are filled with zeros. For example, a buggy-transformed program stripped of all computation would pass this test.

To address this problem, automatic equivalence checking techniques that prove two programs are equivalent have been developed. For instance, focusing on programs with static affine control- and data-flow, prior approaches employ static verification by systematically deriving one-to-one correspondence between the operations in the original and transformed programs and ensuring they satisfy the same data dependences [9, 33, 54]. This has the significant advantage of being independent of the input dataset, yet with the *drawback of requiring both the original and transformed programs to be affine programs*. In other words, any iteration reordering transformation that generates non-affine expressions in the transformed program, such as parametric tiling or numerous abstract syntax tree (AST)-based complementary optimizations, cannot be verified with such an approach. Schordan et al. [46] recently developed an approach to cope with these limitations based on matching the trace of the original and transformed program using rewrite rules and a state transition graph. However, this approach suffers from a major space complexity issue: the traces manipulated are proportional, in size, to the number of operations executed by the program. For example, for an $N \times N$ matrix multiplication code, the trace size is $O(N^3)$.

To date, all proposed solutions to the verification of iteration reordering transformations suffer from at least one major limitation in the supported transformations (e.g., Integer Set Analysis (ISA) [54]), sensitivity to the input dataset (e.g., output difference checking), or space complexity (e.g., CodeThorn [46]), which is the space to store any state (arrays, instruction traces, etc.). In this work, we develop a novel approach to assess the correctness of *arbitrary iteration reordering transformations* on an affine program,

```
for (i=0; i<N; i++)
/*S:*/ A[i] = B[i];
```

(a) Original

```
i=0; do { //N and T are coprime
  A[i] = B[i];
  i = (i+T)%N;
} while (i!=0);
```

(c) Round-robin

```
for (i=0; i<M; i++)
  for(j=idx[i]; j<idx[i+1]; j++)
    A[j] = B[j];
//0=idx[0]<=idx[1]<=...<=idx[M]=N
```

(e) Irregular sectioning

```
for (i=0; i<N/32; i++)
  for (j=0; j<32; j++)
    A[i*32+j] = B[i*32+j];
for(j=(N/32)*32; j<N; j++)
  A[j] = B[j];
```

(b) Constant sectioning

```
for (i=0; i<N/T; i++)
  for (j=0; j<T; j++)
    A[i*T+j] = B[i*T+j];
for(j=(N/T)*T; j<N; j++)
  A[j] = B[j];
```

(d) Parametric sectioning

```
void fn(int lo, int hi) {
  if (lo>hi) return;
  if (lo==hi) A[lo] = B[lo];
  else {
    fn(lo, (lo+hi)/2);
    fn((lo+hi)/2+1, hi); } }
/*Call fn*/ fn(0,N-1);
```

(f) Recursion

Figure 1: Example input program (a) and transformed programs to be verified ((b)–(f)). ISA can verify (b). Our approach can dynamically verify all transformed versions against the input.

including non-affine transformations. The approach is robust to arbitrary input datasets of a given size and only requires space proportional to the program's data space. To achieve this, we design an approach that employs polyhedral data-flow analysis on the affine input program to generate lightweight checker codes that are embedded in the transformed program. The check-equipped transformed code is then run, and information about any violated data dependence or other iteration reordering errors is reported to the user. This enables conclusions about the programs' equivalence for the given input problem size.

We use our PolyCheck tool to verify the correctness of numerous polyhedral compiler optimizations in the Polyhedral Compiler Collection (PoCC) research compiler [7], including non-affine transformations such as parametric tiling and some affine AST-based transformations that are not currently supported by the state-of-the-art affine checker tool ISA [3]. Using PolyCheck, we also verify code generated by Pochoir [50], a DSL-based stencil compiler, and serial execution of Cilk [18], a recursive programming system. By comparing it with the results presented for CodeThorn by Schordan et al. [46], we show that PolyCheck is much more efficient than trace-based equivalence checking schemes. Finally, we demonstrate PolyCheck's ability to identify bugs in the polyhedral compiler PolyOpt/C 0.2.0 [39].

Our approach is orthogonal and complementary to other works that assess the correctness of parallelism transformations (e.g., race detection tools [23, 45]) or out-of-bound array access checks. With this research, we make the following contributions.

- We present the first dynamic bug checker for affine programs transformed by arbitrary iteration reordering transformations using polyhedral analysis at compile-time to create a lightweight checking code to be executed along with the transformed program.
- Our approach addresses the three main limitations of other iteration reordering checkers because it supports arbitrary loop (tiling, interchange, etc.) and non-affine transformations (parametric tiling, etc.), is not sensitive to the input dataset values, and only requires space proportional to that needed by the original program during its execution.
- We demonstrate the effectiveness of our approach in asserting the correctness of Pochoir and Cilk programs and through numerous passes of PoCC, a research polyhedral compiler that combines both affine and non-affine transformations.
- We show that PolyCheck can detect bugs in PolyOpt/C 0.2.0 [39] that were first detected by CodeThorn and is much more efficient than CodeThorn's trace-based equivalence checker.
- We present and evaluate optimizations to the checker that exploit regularities in the program dependence graph.

2. Motivation and Overview

To motivate the need for a verification approach that is robust to arbitrary iteration reordering transformations, we initially use a simple example. We consider the code shown in Figure 1(a), performing a copy from array `B` to array `A`. This code is an affine program with static control- and data-flow. With affine programs, loop bounds, conditionals, and array access expressions only involve affine expressions of the surrounding loop iterators and program parameters [21]. Figures 1(a) through (f) show various transformed versions of this program. Static verification tools checking equivalence between affine programs can verify that the version in Figure 1(b)—tiled with a constant tile size known at code-generation time—is equivalent to the input program. However, they fail to verify the remaining versions. Figure 1(d) shows a code snippet typically generated by parametric tiling techniques that do not require the tile size to be known at code-generation time. Other versions may be manually or automatically generated. The version in Figure 1(e) is non-affine, and its correctness depends on the values of the `idx` array. Recursive versions, such as in Figure 1(f), can be generated for recursive parallelism or cache obliviousness.

The approach we present in this paper can dynamically verify all versions shown in Figure 1. In contrast to static equivalence, we require an instrumented program to be executed. This makes our equivalence property hold for the problem size used when running the checker, a restriction on the proof of equivalence achieved by static checkers. On the other hand, this also enables us to verify arbitrary program transformations. In this work, we support arbitrary *iteration reordering* transformations, irrespective of how the code is generated to implement them. Specifically, we require the transformation to not change the total number of dynamic instances (e.g., N in the figure) of each syntactic statement (e.g., `S`) or the operations performed by the statement. This set of supported transformations includes all loop transformations (e.g., loop tiling [55], index set splitting [30], etc.). Yet, it also includes *any possible syntax to implement the reordering*, including using non-affine expressions in the generated code, AST-based transformations, etc.

We achieve this by constructing a checker to rewrite each statement in the transformed program by operations to check that statement instance. We illustrate the construction of the checker using the iterative Seidel example shown in Figure 2. This example better illustrates the approach than the simpler example in Figure 1(a).

We first statically analyze the affine input (reference) program to determine the read-after-write (true), write-after-read (anti), and write-after-write (output) dependences. Each dependence can be represented as a function (or relation) that takes a statement instance as input and returns the other statement instance associ-

```
1  for t = 0 to  T-1:
2    for i = 1 to N-1:
3      for j = 1 to N-1:
4       S1: A[i][j]=A[i-1][j]+A[i][j-1]
```

Figure 2: Iterative Seidel example based on the language specification in Figure 3. T and N are problem size parameters.

ated with the dependence. For example, in Figure 2, consider the statement instance `S1<t=1,i=5,j=7>`, denoting the instance of statement `S1` executed at the iteration `(t=1,i=5,j=7)`. This statement instance updates the value previously written at `A[5][7]` by `S1<t=0,i=5,j=7>` (output dependence) and reads values of `A[4][7]` and `A[5][6]` (input dependence).

Now consider the operation executed by an unknown statement in the transformed program to be verified:

```
A[8][9] = A[7][9] + A[8][8]
```

We analyze the array locations accessed by an operation in the transformed program to map it to a statement instance in the input program. The preceding operation instance can be mapped to `S1<t=0,i=8,j=9>`, `S1<t=1,i=8,j=9>`, etc., in the input program. In general, an operation encountered in the transformed program can possibly match multiple statement instances in the input program. Given this relationship, we first try to map this operation to a statement instance in the input program that has not already been mapped, and statement instance induced by the mapping satisfies the input program's dependences.

A transformed program is declared to be equivalent to an input program if (a) the statement instances in the transformed program can be mapped in a bijective, or one-to-one, fashion to statement instances in the input program, and (b) each statement instance in such a bijection satisfies the same dependences as in the input program. For arbitrary program transformations, there are numerous ways in which the statement instances of the transformed program can be mapped to those of the input program. Enumerating each mapping to check whether or not it constitutes a dependence-preserving reordering of the statement instances in the input program is prohibitively expensive. We exploit the fact that iteration reordering transformations only change the order in which the statement instances are executed and not the variables written by a given statement instance. We also show that this enables us to check just one mapping to statement instances in the input program.

We can track dependences through the *data space* in terms of the data elements accessed by the statement instances. For each statement instance storing a value in a data element, we use a shadow variable to store the identity of the corresponding input statement instance. For example, if the preceding operation is mapped to `S1<t=1,i=8,j=9>`, the assignment to `A[8][9]` is augmented with:

```
shadow(A[8][9]) = S1<t=1,i=8,j=9>
```

This information can then be used to check the dependence as the program is executed. The checker constructed for this particular statement instance is as follows:

```
assert shadow(A[8][9]) == S1<t=0,i=8,j=9>
assert shadow(A[7][9]) == S1<t=1,i=7,j=9>
assert shadow(A[8][8]) == S1<t=1,i=8,j=8>
shadow(A[8][9]) = S1<t=1,i=8,j=9>
```

where NIL is the initial value of all array locations and variables.

We leverage the fundamental property that if the input program is affine then these true dependences and relationship between statement instances can be represented in a closed form. Together with the shadow variables, this affords compact on-the-fly tracking of the dependences that need to be satisfied. We embed the code to check the equivalence of traces directly in the transformed program. This code can be emitted at compile time because of the input program's affine characteristics.

$$
\begin{array}{lr}
n \in \{M, N, , ..\} & [ProblemSizeParameters] \\
v \in \mathbb{Z} & [Values] \\
l \in \{l_1, l_2, \dots\} & [LoopIndices] \\
A \in \{A_1, A_2, \dots\} & [ArrayNames]
\end{array}
$$

$$
\begin{array}{rl}
a \in \mathit{AffineExpr} ::= & v \mid l \mid v \times l \mid a + a \mid a - a \\
i \in \mathit{ArrayIndex} ::= & a(,a)* \\
lb \in \mathit{LowerBound} ::= & a \mid \textbf{max}(a(,a)+) \\
ub \in \mathit{UpperBound} ::= & a \mid \textbf{min}(a(,a)+) \\
ac \in \mathit{AffineCond} ::= & a > 0 \mid a < 0 \mid a == 0 \mid a! = 0 \\
 & \mid ac \textbf{ and } ac \mid ac \textbf{ or } ac \\
s \in \mathit{Stmts} ::= & A[i] = f_e(v \mid A[i](,v \mid A[i])*) \\
 & \mid s; s \mid \textbf{if } ac : s \\
 & \mid \textbf{for } l = lb \textbf{ to } ub : s \\
p \in \mathit{Programs} ::= & s
\end{array}
$$

Figure 3: Language for affine loop programs. Indentation-based scoping (used for readability) is not shown. f_e is an expression of its arguments.

Note that we only perform static analysis on the input program. Beyond the constraint that the transformed program is an iteration reordered version of the input program, we do not constrain the transformed program. We do not need to perform any analysis on the transformed program. Rather, we only inspect the operations executed by the transformed program, irrespective of how they are generated. In this example, the operation `A[8][9] = A[7][9] + A[8][8]` could have been generated from any code structure with the array input expressions being complex non-affine operations that evaluate to the array index expressions `(8,9)`, `(7,9)`, and `(8,8)`, respectively.

The determination of the one-to-one correspondence requires that the input program statement instance can be computed from the information available in the operations executed by the transformed program. Additional checks are required to detect errors in the transformed program caused by omitted or duplicated statements or operations that do not have a corresponding statement instance in the input program. We will discuss these details and present optimizations to reduce the checking overhead.

3. Background

In this section, we present the basic notation that relates to analysis of affine programs used in the rest of the paper.

To clarify the notation and types of programs we tackle in this paper, we consider the language for affine programs defined in Figure 3. While we use this language for discussion, our implementation handles C programs that conform to this specification. Note that scalars can be treated as one-dimensional arrays of size 1. One significant distinction is the use of a comma-separated list of expressions to represent an array index.[1]

3.1 Integer Sets Notation

Programs with affine data-flow and static control-flow are called static control parts (SCoP) [22, 28], roughly defined as a sequence of statements such that all loop bounds and conditional expressions are affine functions of enclosing loop iterators and variables that are constant during the SCoP execution (whose values may be

[1] This representation of a tuple without surrounding parenthesis ('()' or '[]') allows us to treat function argument lists, array indices, and loop iterator values interchangeably.

unknown at compile time). Affine programs are represented in this work using (a union of) convex sets of integer tuples and (a union of) integer maps. Operations on these structures are readily available in the ISCC calculator [52], which leverages the Integer Set Library [51] to provide operations such as union, intersection, and relation application.

Integer Set The definition of an integer set s is:

$$s = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_m] : c_1 \wedge ... \wedge c_n\}$$

Where $i_1, ..., i_m$ index the m dimensions of the set (noted $\vec{i}$); $p_1, ..., p_p$ are invariant parameters (noted $\vec{p}$); and $c_1, ..., c_n$ are n Presburger formulae, typically in the form of affine inequalities defining constraints on the values of $\vec{i}$.

Integer sets are used to precisely capture the set of runtime instances of statements in affine programs. Each statement S is associated with an iteration vector $\vec{i}_S$ with one component per surrounding loop, and the values $\vec{i}_S$ can take are captured by defining its iteration space $\mathcal{I}_S$. The iteration space of the statement S_1 in Figure 2 is noted $\mathcal{I}_{S_1}$ and is:

$$\mathcal{I}_{S_1} = [T, N] \rightarrow \{S_1[t, i, j] : 0 \leq t < T \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

Relation The definition of an integer relation, or map, $r : \vec{i} \mapsto \vec{j}$ is:

$$r = [p_1, ..., p_p] \rightarrow \{[i_1, ..., i_m] \mapsto [j_1, ..., j_n] : c_1 \wedge ... \wedge c_o\}$$

Relations are used to describe the set of memory locations accessed by statements. In polyhedral programs, array subscripts functions are affine expressions of the loop iterators and parameters. We note them $F_S^{A,i}$ for the i-th access to an array A in a statement S. For example, the statement instances of S_1 that writes array $A[i][j]$ in Figure 2 has $F_S^{A,1} = (i, j)$ and can be represented as:

$$R_{S_1}^{A_1} = \{S_1[t, i, j] \mapsto A[i_1, j_1] : (i_1 = i) \wedge (j_1 = j)\}$$

We note $R_S^{A_i}$ for the i-th read reference map to array A in statement S, and W_S^A as a write reference map to A in S.

Applying relations to sets The apply operation is defined as:

$$(\vec{x} \in s') \iff (\exists \vec{y} \in s \wedge (\vec{y} \mapsto \vec{x}) \in r)$$

where s' is a new set produced by $apply$ of relation r to set s, which can be denoted as $s' = r(s)$.

For instance, the apply operation is used to compute the data space (set of all memory locations accessed) in a loop nest. For example, the write data footprint for array A in Figure 2 is $R_{S_1}^{A_1}(\mathcal{I}_{S_1})$.

Program execution order A schedule is a relation used to specify the execution order of all statement instances. It maps points in the iteration domain to those in an integer set (the set of timestamps). As such, statement instances in the iteration domain are executed following the lexicographic ordering $\prec$ of their associated timestamp. $\prec$ is defined as $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ iff there exists an integer $1 \leq i \leq min(n, m)$ s.t. $(a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1})$ and $a_i < b_i$.

The original program schedule is modeled using $2d + 1$ timestamps, where d is the maximal nesting depth in the program [28]. For example, the schedule of S_1 in Figure 2 is:

$$Sched_{S_1} = \{S_1[t, i, j] \mapsto [0, t, 0, i, 0, j, 0]\}$$

where each odd dimension of the output space is a scalar dimension whose value denotes the lexical AST ordering of the loops surrounding the statement. For statements surrounded by less than d loops, the even schedule components associated with the missing loops is set to 0. However, we use a special convention where the last component of the schedule corresponds to a unique identification for the statement (e.g., 1 for S_1) instead of the lexical AST order of the statement in this loop nest.

3.2 Polyhedral Dependences

In the polyhedral model, dependences for affine computations (such as flow dependence and output dependence) can be precisely calculated and expressed as relations from a source iteration to a target iteration in the iteration space [27, 42, 43].

Polyhedral dependences can be obtained using tools such as ISL [4]. For example, one flow dependence in the example code in Figure 2 is shown as follows:

$$Flow = [T, N] \rightarrow \{S_1[t, i, j] \mapsto S_1[t, i, 1 + j] : 0 \leq t < T \wedge 1 \leq i < N \wedge 1 \leq j < N - 1\}$$

The flow relation shows that a flow dependence exists between iteration (t, i, j) and $(t, i, j + 1)$ for statement S_1. The value of $A[i][j]$ written by statement S_1 at iteration (t, i, j) is used later at iteration $(t, i, j + 1)$ within the iteration space.

Note that we only consider *exact dependences*, meaning we only consider the *last write* of source iteration for any dependence pair from source to target in write-after-write (WAW or output) and read-after-write (RAW or flow) dependences.

4. Verifying Transformations on Affine Programs

In this section, we present our approach for verifying that the transformed program has been obtained by a valid reordering of the iterations of the input program. This involves identifying a one-to-one correspondence between the statement instances in the input and the transformed program, such that, for each statement instance in the input program, the corresponding statement instance in the transformed program satisfies the same dependences. In general, verifying equivalence requires comparing traces of the two programs to derive a graph isomorphism—an expensive task. We show that transformations restricted to iteration reordering can be verified at a much lower cost.

PolyCheck specification The affine specification of a transformed program, to be used by PolyCheck is provided as follows:

```
/*@ polycheck start spec
 <affine input program>
*/
<transformed program>
/*@ polycheck end
*/
```

Note that all operations reachable from the *segment* enclosed in the transformed program are expected to be available at instrumentation time. In other words, any function that could be transitively called from `<transformed program>` in the above code shall be instrumented.

4.1 Algorithm A: A General Algorithm for all Affine Input Programs

Our approach to verification combines two stages. First, we analyze the *input* affine program to build a series of functions used to extract properties and assertion values from an executed operation in the *transformed* program. Second, we instrument the transformed program to call these functions for each executed operation, checking if the operation's runtime properties match the expected values computed by these functions. In a nutshell, the process is as follows. Assume the transformed program so far is valid, and an operation o attempts to write to memory location l. The last valid iteration `S<itrv>`[2] that wrote l is stored in `shadow(l)`. From there, we

[2] Throughout the paper, we use multi-character identifiers that end in v, e.g. `itrv`, to denote vectors.

can determine what should be S<itr2v>, the next iteration writing to l, by evaluating a function we built via static analyis of the input program that provides the $NextWriter$ iteration. We can then determine what memory location is accessed by S<itr2v> and using an analogous process what iteration S<itrXv> was the producing iteration for each memory location accessed by S<itr2v>. This information is then checked against the runtime information of $\circ$: the observed shadow value for all its operands and the memory addresses accessed must all match with the values associated with S<itr2v>. If any of these values do not match, then $\circ$ is an invalid operation, and the transformed program does not match the input program. If all operations $\circ$ are valid and every $\circ$ was matched with exactly one iteration S<itrv> from the input program, the transformed progam matches the input program.

In the following, we first define the various functions extracted via static analysis to compute the $FirstWriter$, $NextWriter$, etc., instances for a memory location in Sec. 4.1.1, before detailing the runtime checking algorithm in Sec. 4.1.2.

4.1.1 Compile-time analysis of the input program

The first stage of analysis involves producing functions that can be evaluated at runtime in the tranformed code. These functions use various integer set/map operations, such as union ($\cup$), computing the lexicographic minimum ($lexmin$) and maximum ($lexmax$), and application of a map to a set ($M(S)$).

FirstWriter The function $\vec{t} = FirstWriter(A, \vec{w})$ computes which instance $\vec{t}$ is the first one in the input program to write to a specific memory location $\vec{w}$ for array A. To compute this function, we proceed by computing for each array A written in the program a function that returns the timestamp of the first iteration writing to an arbitrary memory location.

We model an arbitrary memory location in array A as a parametric point in a set:

$$Point_A = [\vec{W}] \rightarrow \{[\vec{w}] : \vec{w} = \vec{W}\}$$

where $\vec{w}$ and $\vec{W}$ have the same dimensionality as the array A (e.g., a 2D set for a two-dimensional array). We then express the set of instances that write to A as a map from the array index accessed to the iteration accessing it. For array A written by statement S it is:

$$M_S^A = [\vec{P}] \rightarrow \{[\vec{w}] \mapsto [\vec{i}] : W_S^A(\vec{i}) = \vec{w} \wedge \vec{i} \in \mathcal{I}_S\}$$

where $\vec{P}$ is the vector of program parameters. Therefore, the timestamp associated with the instance writing to an arbitrary but unique memory location $\vec{w}$ is:

$$T = Sched_S(M_S^A(Point_A))$$

Finally, given $\mathcal{S}$ the set of statements in the input program, one can build the first instance across the whole program accessing a particular location $\vec{w}$: $FirstWriter(A, \vec{w}) =$

$$lexmin\left(\bigcup_{S \in \mathcal{S}} \left(Sched_S(M_S^A(Point_A)\right)\right)$$

We remark that the preceding sets and relations, including the $lexmin$ operation, can be seamlessly computed using, for instance, the ISCC calculator. The $lexmin$ returned is an expression (typically a tree of expressions, where leaves are possible $lexmin$ values and nodes in the tree are conditions on the numerical values of $\vec{w}$ and $\vec{P}$). To build the function FirstWriter(A,w), we simply translate this tree of expressions to C code. The process is repeated for all written arrays in the program, and each is embedded in the function's code so the function selects the appropriate $lexmin$ expression tree to evaluate as a function of the array name considered.

NextWriter The function $\vec{t} = NextWriter(\vec{t}_{prev}, A, \vec{w})$ computes which is the instance $\vec{t}$ in the input program writing to the location $\vec{w}$ of A executing immediately after $\vec{t}_{prev}$ wrote to the same memory location $\vec{w}$. To build this function, we employ methods similar to those for $FirstWriter$. We first model an arbitrary iteration of a program as a parametric point in a set:

$$Iter1 = [\vec{T}] \rightarrow \{[\vec{t}] : \vec{t} = \vec{T}\}$$

where $\vec{t}$ and $\vec{T}$ have $2d+1$ components, the number of dimensions of a timestamp (i.e., output dimensions of scheduling functions). To capture an iteration $\vec{t}$ immediately following another iteration $\vec{t}_{prev}$, we use a slight extension of the definition of M_S^A to add input dimensions and capture $\vec{t}_{prev}$ as follows:

$$N_S^A = [\vec{P}] \quad \rightarrow \quad \{[\vec{w}, \vec{t}_{prev}] \mapsto [\vec{i}] : W_S^A(\vec{i}) = \vec{w} \ \wedge \ \vec{i} \in \mathcal{I}_{S_1} \\ \wedge \ \vec{t}_{prev} \prec Sched_S(\vec{i})\}$$

Finally $NextWriter(\vec{t}_{prev}, A, \vec{w})$ is built using a similar expression as for $FirstWriter(A, \vec{W})$, simply substituting $M_S^{A,j}$ by $N_S^{A,j}$ and $Point_A$ by $(Iter1, Point_A)$.

WriterBeforeRead $\vec{t} = WriterBeforeRead(\vec{t}_{read}, A, \vec{w})$ computes the instance $\vec{t}$ that last wrote to a location $\vec{w}$ of array A read by iteration $\vec{t}_{read}$ in the input program. This is essential to make sure data dependences are preserved in the programs. Writes must all occur in the input order, and read values must contain the same value as in the input program when a particular instance executes. This function is computed in a manner analogous to $NextWriter$, except instead of stating $\vec{t}_{prev} \prec Sched_S(\vec{i})$ we state $Sched_S(\vec{i}) \prec \vec{t}_{read}$, and compute the $lexmax$ of the problem instead of the $lexmin$ to have the instance immediately preceding $\vec{t}_{read}$ writing to a location $\vec{w}$. Note that if $A(\vec{w})$ is an input location not yet overwritten by any statement instance (e.g., it is live-in data), then $\vec{t}$ is set to Init<0>.

LastWriter The function $\vec{t} = LastWriter(A, \vec{w})$ computes the instance $\vec{t}$ that is last to write to a memory location $\vec{w}$ in the array A. This is simply the converse of the $FirstWriter$ function, which is formulated identically except the $lexmax$ instead of the $lexmin$ is used to find $\vec{t}$.

WriteSet and InputSet We conclude by defining two convenience sets, namely the WriteSet and InputSet, used to initialize the checking procedure.

The WriteSet is the set of array locations written to by the input program, we initialize their shadow to NIL. This corresponds to the data space built from the union of the data spaces induced by all write references in the program. Its expression, for an array A, is:

$$WriteSet(A) = \bigcup_{S \in \mathcal{S}} W_S^A(\mathcal{I}_S)$$

The InputSet is the set of data being read before being written, or being read-only. It is used to initialize the last writing instance to a default starting value Init<0>. It is assembled by first building the set of all memory locations being read. For the array A, it is: $ReadSet(A) = \bigcup_{S \in \mathcal{S}} \left(\bigcup_i R_S^{A_i}(\mathcal{I}_S)\right)$, where i ranges to cover each read access function to A in S.

We then prune this set from all the memory locations being written before being read, which is obtained by applying $WriterBeforeRead$ on each first-read instance per memory location, keeping only locations where it returns Init<0>. The set of first-read instances is computed in a manner analogous to the $FirstWriter$, except considering read access functions.

4.1.2 Compile-time analysis of the transformed program

We present the notation used in the algorithm in a manner closer to the AST form that occurs in program analysis. Each notation that follows can be associated to the notation used in the previous section for the various functions. For example $\vec{w}$ in A is `A[wv]`.

Notations An array location is denoted by the array label and index vector (e.g., `A[idxv]`), where `idxv` is a tuple of length equal to the dimensionality of array `A`.

A statement instance is denoted by the statement label and the iteration vector (e.g., `S<itrv>`).

Arrays are indexed using array reference functions. Given an iteration vector as input, an array reference function returns the index vector to index into the array. The array reference function to compute the array index written by statement `S` is denoted by `S.w()`. The array index written by a statement instance `S<itrv>` is computed as `S.w(itrv)`.

The list of array reference functions to compute indices used in read array references in statement `S` is denoted by `S.r`. The array indices read by a statement instance `S<itrv>` is computed as `S.r[0](itrv)`, `S.r[1](itrv)`, etc.

The verification algorithm employs shadow state for every array location of interest. The shadow state associated with location `A[idxv]` is denoted by `shadow(A[idxv])`.

For every operation `o` in the transformed program, the arrays written and read by it are noted by `o.Aw` and the list `o.Ar`, respectively. The array indices for the writes and reads are denoted by `o.wv` and `o.rv()`.

Verification algorithm First, a simple static analysis of the transformed progam is performed to identify each statement that can write to any array written by the input program. Each operation performed by these statements is marked to generate runtime check operations. We note `ts(o)` as the statement in the transformed program that generates this operation `o`. The function `CandidateInputStmts(ts)` returns the set of all statements in the input program whose syntactic structure matches that of `ts`. We then analyze and instrument all code reachable from the transformed program segment to be analyzed.

Figure 4 depicts the runtime checking algorithm. For simplicity, we represent function arguments as an object that contains all of the necessary parameters to evaluate the functions defined in Sec. 4.1.1. The *WriterBeforeRead* is noted `RAW`.

```
1  @initialize:
2    //executed before starting execution of
       transformed program
3    for w in WriteSet:
4      shadow(w) = NIL
5    for i in InputSet:
6      shadow(i) = Init<0>
7
8  def input_statement_instance(o):
9    assert shadow(o.Aw[o.wv]) exists
10   if shadow(o.Aw[o.wv]) == NIL:
11     S<itrv> = FirstWriter(o.Aw[o.wv])
12   else:
13     S<itrv> = NextWriter(shadow(o.Aw[o.xv]),
       o.Aw[o.wv])
14   assert S<itrv> exists //valid iteration
15   assert S in CandidateInputStmts(ts(o))
16   return S<itrv>
17
18 @verify(o):
19   //executed for every operation encountered
        in the transformed program
20   S<itrv> = input_statement_instance(o)
21   for i in 0 .. |S.r|-1:
22     assert S.r[i](itrv) == o.rv[i]
23     assert RAW(S,i)(itrv) == shadow(o.Ar[i][
       o.rv[i]])
24   shadow(o.Aw[o.wv]) = S<itrv>
25
26 @terminate:
27   //executed after execution the transformed
        program
28   for w in WriteSet:
29     assert shadow(w) == LastWriter(w)
30   report SUCCESS
```

Figure 4: Runtime procedure to check every operation encountered in the transformed program for algorithm A.

4.2 Algorithm B: A Version-Number based Algorithm

The algorithm in Figure 4 enables the verification of program transformations made on arbitrary input affine programs. This generality comes at a runtime overhead cost. For example, functions which need to be evaluated for each operation, especially *NextWriter* and `RAW`, have been generated at compile time as a solution to a parametric lexicographic minimum/maximum on a union of sets. In other words, possibly many if conditionals necessarily need to be evaluated for each instance. Also, although the space overhead remains linear in the input dataset size, storing multidimensional instance vectors in programs nested by d loops requires $2d+1$ integer attributes per memory location accessed in the input program. To overcome these limitations, we present a slightly different technique that requires storing only a single integer per memory location and, in most cases, does not require the evaluation of deep conditionals for each operation. We achieve this by restricting the class of affine programs handled to those where *the iteration vector of an operation can be computed by solving linear equations using the index value of the memory locations accessed by the operation*. We also do not store the last instance that have written in this location, but instead the *number of instances which previously wrote to this location*.

Compile-time analysis of the input program The analysis extracts the `WriteSet` and `InputSet` similarly to the general approach, as well as `LB(S)`, the iteration domain of the statement S (i.e., $\mathcal{I}_S$).

A `VersionNumber(S)` function is built such that, given a instance `S<itrv>` of statement S, it computes how many instances previously wrote to this location in the input program and returns this value+1. This is computed by forming the set of such an instance using similar concepts as the general algorithm and building a counting polynomial for the resulting parametric set. We use Barvinok's techique [53] available in ISCC. The function `RAWv(S,i)` returns the `VersionNumber` of `RAW(S,i)`.

The function `NumWrites(w)` returns the number of statement instances that write to array location `w` in the input program by building the counting polynomial on `WriteSet(w)`.

The function `LIN(S)` produces a list of linear functions that consists of:

- Affine array reference functions S.w and each function in `S.r`
- For all i: `RAWv(S,i)` if `RAWv(S,i)` is affine
- `VersionNumber(S)` if `VersionNumber(S)` is affine.

We ensure that the matrix associated with `LIN(S)` is full rank (rank equal to loop nest depth of statement S) for all S. If not, the version-number based scheme cannot be used. Instead, the algorithm shown in Figure 4 must be employed.

Finally, the function `RHS(S)` returns a vector of loop indices, expressed in terms of array indices and version numbers, that forms

Table 1: FirstWriter to array A in iterative Seidel

	0	1	2	3
0	–	–	–	–
1	–	S[0,1,1]	S[0,1,2]	S[0,1,3]
2	–	S[0,2,1]	S[0,2,2]	S[0,2,3]
3	–	S[0,3,1]	S[0,3,2]	S[0,3,3]

the right-hand side of the equation to solve for x in `LIN(S).xv = RHS(S)`.

Verification algorithm The compile-time analysis of the transformed program is performed in a manner identical to the general algorithm. Figure 5 gives the pseudocode that instruments the transformed program for this scheme.

```
1  @initialize:
2    //executed before starting execution of
       transformed program
3    for w in WriteSet:
4      shadow(w) = 0
5    for i in InputSet:
6      shadow(i) = 0
7
8  def input_statement_instance(o):
9    assert shadow(o.Aw[o.wv]) exists #writing
       to a location in WriteSet
10   for S in CandidateInputStmts(ts(o)):
11     Solve for x in LIN(S) . xv = RHS(S)(o.w,
        o.rv[1], ..) s.t. xv in LB(S)
12     if xv is found:
13       //check below is redundant if
       VersionNumber(S) is a linear function
14       if VersionNumber(S)(xv)==shadow(o.Aw[o
       .wv]):
15         return S<xv>
16   assert false //does not match any input
       statement instance
17
18 @verify(o):
19   //executed for every operation encountered
       in the transformed program
20   S<itr> =  input_statement_instance(o)
21   for i in 0 .. |S.r|-1:
22     if RAWv(S,i) not in LIN(S): //non-linear
        component
23       assert shadow(o.Ar[i][o.rv[i]]) exists
24       assert RAWv(S,i)(itr) == shadow(o.Ar[i
       ][o.rv[i]])
25   shadow(o.Aw[o.wv]) += 1
26
27 @terminate:
28   //executed after execution the transformed
        program
29   for w in WriteSet:
30     assert shadow(w) == NumWrites(w)
31   report SUCCESS
```

Figure 5: Runtime procedure to check every operation encountered in the transformed program for algorithm B.

Theorem 1. *Algorithm A (resp. B) terminates without error if and only if the operations that were verified in the transformed program execution correspond to a dependence-preserving reordering of the statement instances in the input program.*

Proof. Proof is presented in [10]. □

```
1  seidel_rec(t,ilo,ihi,jlo,jhi,T,N,A[N][N]):
2    if ilo>ihi || jlo>jhi: return
3    if ilo==ihi && jlo==jhi:
4      A[ilo,jlo] = A[ilo-1,jlo] + A[ilo,jlo-1]
5    else:
6      seidel_rec(t,ilo,⌊(ilo+ihi)/2⌋,
7        jlo,⌊(jlo+jhi)/2⌋,T,N,A) //Top-Left
8      seidel_rec(t,ilo,⌊(ilo+ihi)/2⌋,
9        ⌊(jlo+jhi)/2⌋+1,jhi,T,N,A) //Top-Right
10     seidel_rec(t,⌊(ilo+ihi)/2⌋+1,ihi,
11     jlo,⌊(jlo+jhi)/2⌋,T,N,A) //Bottom-Left
12     seidel_rec(t,⌊(ilo+ihi)/2⌋+1,ihi,
13     ⌊(jlo+jhi)/2⌋+1,jhi,T,N,A) //Bottom-Right
14   if ilo==1 && ihi==N-1 && jlo==1 && jhi==N
       -1 && t<T:
15     seidel_rec(t+1,ilo,ihi,jlo,jhi,T,N,A)
             // Next time step
```

Figure 6: Recursive implementation of Seidel. Invoked as `seidel_rec(0,1,3,1,3,2,4,A)`.

4.3 Illustrative Example: Seidel

We provide an intuition of the algorithms' operation using the Seidel example. The key idea behind the developed approach (common to both Algorithms A and B) is to perform on-the-fly matching of each operation (executed statement instance) of the transformed program being checked with some statement instance in the execution of the input affine program. The test program uses a shadow variable for each data variable, and as each operation of the tested program is successfully matched with a statement instance in the input program, the information is stored in the shadow variable for the data element written by the operation. The stored information either directly (Algorithm A) or indirectly (Algorithm B) enables identification of the matched statement instance in the input program's execution. If the on-the-fly matching process succeeds, it essentially identifies a valid dependence-preserving bijection between the input program's statement instances and the sequence of operations executed by the transformed program. In the dynamic matching process, if we are unable to successfully match any operation of the transformed program being checked, it means there is an error in the transformed program: its sequence of operations is provably not equivalent to any dependence-preserving reordering of the input program's statement instances.

We use an example to illustrate the verification approach. Figure 2 shows code for a 2D iterative stencil computation. The code sweeps through a 2D array `A`, row by row, updating element `A[i][j]` using the values of two neighboring elements, `A[i-1][j]` and `A[i][j-1]` (inner loops over `i,j`). The sweep over the 2D array is repeated for multiple time steps (outermost `t` loop). Figure 6 shows recursive code that performs the same computation. It splits the spatial domain of the sweep into four quadrants and recursively invokes sweeps on them, in the order top-left, top-right, bottom-left, and bottom-right. The base case performs the stencil operation on a single element.

Table 2 shows the sequence of 18 statement instances for the execution of the iterative Seidel code for `N=4,T=2`. For each statement instance, we see the written element of A, the two read elements of A, along with some additional information about data dependences that can be computed in a straightforward way in a polyhedral compiler framework for any affine program: 1) the latest preceding statement instance (PrevW) that wrote to any given data element, 2) the earliest future statement instance (NextW) that will write to any given data element. Shadow variables for input

Table 2: Statement instances executed by iterative Seidel and related relations

Stmt	Write	NextW	Read1	PrevW	Read2	PrevW
S<0,1,1>	A[1,1]	S<1,1,1>	A[0,1]	–	A[1,0]	–
S<0,1,2>	A[1,2]	S<1,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
S<0,1,3>	A[1,3]	S<1,1,3>	A[0,3]	–	A[1,2]	S<0,1,2>
S<0,2,1>	A[2,1]	S<1,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–
S<0,2,2>	A[2,2]	S<1,2,2>	A[1,2]	S<0,1,2>	A[2,1]	S<0,2,1>
S<0,2,3>	A[2,3]	S<1,2,3>	A[1,3]	S<0,1,3>	A[2,2]	S<0,2,2>
S<0,3,1>	A[3,1]	S<1,3,1>	A[2,1]	S<0,2,1>	A[3,0]	–
S<0,3,2>	A[3,2]	S<1,3,2>	A[2,2]	S<0,2,2>	A[3,1]	S<0,3,1>
S<0,3,3>	A[3,3]	S<1,3,3>	A[2,3]	S<0,2,3>	A[3,2]	S<0,3,2>
S<1,1,1>	A[1,1]	–	A[0,1]	–	A[1,0]	–
S<1,1,2>	A[1,2]	–	A[0,2]	–	A[1,1]	S<1,1,1>
S<1,1,3>	A[1,3]	–	A[0,3]	–	A[1,2]	S<1,1,2>
S<1,2,1>	A[2,1]	–	A[1,1]	S<1,1,1>	A[2,0]	–
S<1,2,2>	A[2,2]	–	A[1,2]	S<1,1,2>	A[2,1]	S<1,2,1>
S<1,2,3>	A[2,3]	–	A[1,3]	S<1,1,3>	A[2,2]	S<1,2,2>
S<1,3,1>	A[3,1]	–	A[2,1]	S<1,2,1>	A[3,0]	–
S<1,3,2>	A[3,2]	–	A[2,2]	S<1,2,2>	A[3,1]	S<1,3,1>
S<1,3,3>	A[3,3]	–	A[2,3]	S<1,2,3>	A[3,2]	S<1,3,2>

Table 3: Sequence of operations for recursive Seidel; "Sdw" abbreviates shadow in the column headers

Write	Sdw_old	Sdw_new /Match	Read_1	Shadow	Read_2	Shadow
A[1,1]	–	S<0,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	–	S<0,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
A[2,1]	–	S<0,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–
A[2,2]	–	S<0,2,2>	A[1,2]	S<0,1,2>	A[2,1]	S<0,2,1>
A[1,3]	–	S<0,1,3>	A[0,3]	–	A[1,2]	S<0,1,2>
A[2,3]	–	S<0,2,3>	A[1,3]	S<0,1,3>	A[2,2]	S<0,2,2>
A[3,1]	–	S<0,3,1>	A[2,1]	S<0,2,1>	A[3,0]	–
A[3,2]	–	S<0,3,2>	A[2,2]	S<0,2,2>	A[3,1]	S<0,3,1>
A[3,3]	–	S<0,3,3>	A[2,3]	S<0,2,3>	A[3,2]	S<0,3,2>
A[1,1]	S<0,1,1>	S<1,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	S<0,1,2>	S<1,1,2>	A[0,2]	–	A[1,1]	S<1,1,1>
A[2,1]	S<0,2,1>	S<1,2,1>	A[1,1]	S<1,1,1>	A[2,0]	–
A[2,2]	S<0,2,2>	S<1,2,2>	A[1,2]	S<1,1,2>	A[2,1]	S<1,2,1>
A[1,3]	S<0,1,3>	S<1,1,3>	A[0,3]	–	A[1,2]	S<1,1,2>
A[2,3]	S<0,2,3>	S<1,2,3>	A[1,3]	S<1,1,3>	A[2,2]	S<1,2,2>
A[3,1]	S<0,3,1>	S<1,3,1>	A[2,1]	S<1,2,1>	A[3,0]	–
A[3,2]	S<0,3,2>	S<1,3,2>	A[2,2]	S<1,2,2>	A[3,1]	S<1,3,1>
A[3,3]	S<0,3,3>	S<1,3,3>	A[2,3]	S<1,2,3>	A[3,2]	S<1,3,2>

and output data elements before any assignment are both shown as null ('–'). Consider, for example, the two statement instances that write to `A[2][1]`: `S<0,2,1>` and `S<1,2,1>`. In each of the two statement instances, the read data elements are `A[1][1]` and `A[2][0]`. Of these two, `A[2][0]` is a boundary element that is not written within the execution of the code. Therefore, the "previous writer" is null. However, `A[1][1]` is modified in the code by statement instances `S<0,1,1>` and `S<1,1,1>`. The table row for statement instance `S<0,2,1>` lists `S<0,1,1>` as the previous writer of the read data element `A[1][1]`, while the row for `S<1,2,1>` lists `S<1,1,1>` as the previous writer of `A[1][1]`. The next writer for the written data element `A[2][1]` is `S<1,2,1>` for statement instance `S<0,2,1>` and null for statement instance `S<1,2,1>`. Table 1 displays the first statement instance that writes into each element of array `A`. The top and left boundary elements are only read but not written, so they have null as their "first writer."

Table 3 shows the sequence of 18 operations executed by the checker for the recursive Seidel code. The table entries include the read and written elements of A for each instance, along with the values of the shadow variables associated with each data element. The shadow variables are all initialized to null. Consider the first operation executed by the recursive version: it writes into `A[1][1]`, whose shadow is currently null. The first writer of `A[1][1]` in the input program is determined (shown in Table 1, but actually dynamically generated, as shown later) to be `S<0,1,1>`. Thus, as long as the operands and their previous writers also match, this operation can get matched with that statement instance of the input program. The previous writers of the operands `A[0][1]` and `A[1][0]` for `S<0,1,1>` are both null, matching the null `shadow(A[0][1])` and `shadow(A[1][0])` in the checker program for the recursive version. `Shadow(A[1][1])` is set to the successfully matched input program statement instance `S<0,1,1>`. Next, consider the second operation in the execution of the recursive version that writes into `A[1][1]`. To match this operation to a statement instance of the input program, the shadow variable value `S<0,1,1>` is used, and the next-writer in the input program is determined to be `S<1,1,1>`. The read data elements and their previous writers (null) match the shadow variables in the executing checker program, implying a match. In a similar manner, all operations in the recursive execution can be matched one-to-one with a statement instance of the iterative version.

Table 4: Sample operations for recursive Seidel with error; "Sdw" abbreviates shadow in the column headers

Write	Sdw_old	Sdw_new /Match	Read_1	Shadow	Read_2	Shadow
A[1,1]	–	S<0,1,1>	A[0,1]	–	A[1,0]	–
A[1,2]	–	S<0,1,2>	A[0,2]	–	A[1,1]	S<0,1,1>
A[2,2]	–	S<0,2,2>	A[1,2]	S<0,1,2>	A[2,1]	–
A[2,1]	–	S<0,2,1>	A[1,1]	S<0,1,1>	A[2,0]	–

To illustrate the verification process further, consider what happens when the tested program is not equivalent to the input program. For example, for the recursive Seidel version, assume that the last two of the four recursive calls to top-left, top-right, bottom-left, and bottom-right got swapped by mistake, i.e., the sequence of calls instead becomes top-left, top-right, *bottom-right, bottom-left*. Table 4 shows the first few operations executed, along with shadow variable information. The on-the-fly matching is successful for the first two operations (writing into `A[1][1]` and `A[1][2]`, respectively) but will fail for the third operation, which writes into `A[2][2]`. Because the shadow variable has a null value, the matching process will use first-writer information for `A[2][2]` and attempt a match with the iterative program's statement instance `S<0,2,2>`. Of the two read data elements, we get a match for `A[1][2]` (the previous writer in the iterative program statement instance, `S<0,1,2>`, matches the shadow variable in the test program) but not for A[2][1]. The shadow variable for `A[2][1]` is null as `A[2][1]` has not yet been written in the test program so far. However, the previous writer for `A[2][1]` in the matched statement instance is `S<0,2,1>`. This mismatch means the test program cannot be equivalent to the input program.

Figures 7 and 8 show the checker code generated for recursive Seidel code for using Algorithms A and B, respectively. Figure 7 first shows initialization code that sets all shadow variables to null followed by the verification code inserted after the statement in line 4 of the recursive Seidel code (Figure 6)—the operation performed at the base case of the recursion. The checker code encodes the previously described matching process using the example traces. Finally, an epilog code verifies that all operations were performed by the transformed program. This is done by ensuring the final shadow

```
1  @initialize: //initialize shadow variables
2    for (i,j) in (0,0) to (N-1,N-1):
3     shadow(A[i][j]) = NIL
4  @verify: //inserted after line 4 in
       seidel_rec
5    i = ilo
6    j = jlo
7    assert i-1==ilo-1 //check 1st index and
8    assert j==jlo     //2nd index in A[ilo-1][
       jlo]
9    assert i==ilo     //check 1st index and
10   assert j-1==jlo-1 //2nd index in A[ilo][
       jlo-1]
11   if shadow(A[i][j])==NIL: // Write
12     assert FirstWriter(A[i][j])==S<t,i,j>
13   else:
14     assert shadow(A[i][j])==S<t-1,i,j>
15   if shadow(A[i-1][j])!=NIL: // Read
16     assert shadow(A[i-1][j])==S<t,i-1,j>
17   if shadow(A[i][j-1])!=NIL: // Read
18     assert shadow(A[i][j-1])==S<t,i,j-1>
19   shadow(A[i][j]) = S<t,i,j>
20 @terminate: //epilog to ensure completeness
21   for (i,j) in (1,1) to (N-1,N-1):
22     assert shadow(A[i][j]) == S<T-1,i,j>
```

Figure 7: The checker code inserted by Algorithm A after line 4 in the recursive Seidel implementation (Figure 6). Note that the compiler can optimize away the first four assert statements.

```
1  @initialize://initialize shadow variables
2    for (i,j) in (0,0) to (N-1,N-1):
3     shadow(A[i][j]) = 0
4  @verify: //inserted after line 4 in
       seidel_rec
5    Determine (t,i,j) that satisfy the
       following inequalities and equalities:
6      0<=t<T and 0<j<N and 0<j<N //loop bounds
7      i-1=ilo-1 //check 1st index and
8      j  =jlo   //2nd index in A[ilo-1][jlo]
9      i  =ilo   //check 1st index and
10     j-1=jlo-1 //2nd index in A[ilo][jlo-1]
11     //shadow values
12     shadow(A[i][j])  =(i>0 && j>0) ? t   : 0
13     shadow(A[i-1][j])=(i>1 && j>0) ? t+1 : 0
14     shadow(A[i][j-1])=(i>0 && j>1) ? t+1 : 0
15   if no (t,i,j) found: report error
16   shadow(A[i][j]) += 1
17 @terminate://epilog to ensure completeness
18   for (i,j) in (1,1) to (N-1,N-1):
19     assert shadow(A[i][j]) == T
```

Figure 8: The checker code inserted by Algorithm B after line 4 in the recursive Seidel implementation (Figure 6).

variables match the analytically computable last writers for the input affine program. Figure 8 shows the checker code for using the optimized approach of Algorithm B, which is applicable for this example. Instead of keeping full details of matched statement instances in shadow variables, a compact version counter is stored in each shadow variable, and the matching process involves comparing the values in the version counters with analytically computable write counts for each data element in the affine input program.

4.4 Time and Space Complexity

For each operation to be checked by the transformed program, the number of actions taken by the checker is proportional to the number of array references and loop nesting depth of each statement. Specifically, computing the corresponding statement instance requires time proportional to the loop nesting depth. Checking each array reference requires computing the array index from an iterator value with the cost proportional to that of computing the array references in the input program. Note that `LIN(S)` is known when analyzing the input affine program. Rather than solve the system of equations at runtime, we compute the (pseudo-)inverse for `LIN(S)` at compile time and only perform a matrix-vector product at runtime. In general, the cost of checking each operation using Algorithm A is proportional to the loop nesting depth. Given the loop nesting depth is usually small, the checker cost is on the same order as executing the input program. Algorithm B might undertake several mappings for each operation—in the worst case, attempting a mapping with every statement in the input program. Therefore, the worst case cost of checking each operation using Algorithm B is proportional to the loop nesting depth times the number of statements in the input program. We only employ Algorithm B when the number of candidate input statements for any given statement in the transformed program is small.

The operations in the transformed program are processed in a streaming fashion as they are generated with no storage of past operations. The only significant space required by the checker is the shadow variable associated with each location read and written by the input program. Each shadow variable holds a constant-sized object (statement instance object or version number). Therefore, the algorithm's total space complexity is the same as that of the input affine program, and is independent of the number of statement instances executed by the input or the transformed program.

5. Scope of Applicability, Enhancements, and Limitations

The following describes a few key performance enhancements for the PolyCheck implementation.

Optimized checking of full tiles As discussed, we add several instructions for each statement in the transformed program. This overhead can be significantly improved for a common class of transformed programs. Specifically, tiled programs group statement instances into tiles to improve data locality. While the bounds for these tiles can be constants or influenced by a tile-size parameter, the code within a tile itself is often an affine program. In such scenarios, we employ index set splitting to isolate tiles that depend only on linear combinations of problem- and tile-size parameters. We outline such tiles and statically analyze them to identify the incoming definitions—array locations last written elsewhere in the program and read within this tile. Statements with these definitions are verified in full. Other statements that only read array locations written by statement instances within the same tile (intra-tile dependences) are statically verified. Each statement is replaced with an increment of the version number of the array element written by that statement. This ensures that version counts are still maintained to enable continued verification of the remaining program.

Delayed detection and vectorization If the termination and error reporting criteria can be relaxed, the verification performance can be improved. Checking and aborting on each verification introduces several branches and leads to poor performance. Most checks compare the expected version number for the input of a dependence with the version number observed. Thus, we replace the check:

`assert a==b` with `checksum |= (a^b)`

At the end of the program, we verify the checksum:

`assert checksum==0`

At the end of the iteration, a non-zero checksum value indicates detection of an error in the transformed program. The bitwise or operation ensures that any error detected (using the xor operation)

results in a bit being set and never subsequently unset. Therefore, the checksum is exact in that any error detected during the checker execution is eventually reported. When the checks are enclosed in loops with statically known loop bounds in the transformed code, we place the scalar checksum variable with an array to make the code amenable to vectorization.

Localizing bugs In addition to detecting errors, PolyCheck strives to isolate the offending statement instance and report the reason for failed verification. For each offending operation, we report the operation and information on the statement instance of the transformed program. When a given operation cannot be mapped to a dependence-preserving statement instance s in the input program that writes to the same location, we attempt to find alternative possible mappings for the same operation to other non-dependence-preserving statement instances. If the operation maps to a statement instance s' that lexicographically precedes s, we report the offending operation as a duplicate of an earlier operation generated by s. If the operation maps to a statement instance s' that lexicographically follows s, we report the offending operation as having executed too soon, violating a write-after-write dependence. When no valid mapping can be found for any version number, we report it as an invalid operation. When an operation is found to violate a dependence, we report the observed and anticipated statement instances in the input program. This allows the programmer to locate the offending statement instance in the transformed program, the corresponding statements in the input program, and the cause for the failed verification.

Problem-size parameters When the array access functions depend on problem-size parameters, information about them must be extracted from the transformed program as well. We support two approaches. First, the user can provide the problem size as a simple annotation to the verifier. This is expected to be one or a small number of annotations, one per parameter, and is not an onerous requirement. Second, we can treat the parameters as variables and solve for them as we solve for the loop indices. Assuming that there are sufficient linearly independent constraints involving the parameter of interest, we compute the parameters the first time they are encountered. In subsequent checks, the discovered parameters are treated as constants. In general, problem-size parameters can be discovered by relating properties of a full polyhedron—number of integer points, last write to a given location, extremes on the array locations read/written, etc.—and relating them to the values observed when running the transformed program.

While PolyCheck can handle numerous program variants, it is still restricted to iteration reordering transformations. The scope of transformations and programs that PolyCheck can handle is discussed further in the following.

Aliasing Affine dependence analysis assumes that two locations `A1[itr1v]` and `A2[itr2v]` are identical (refer to the same location) only if `A1=A2` and `itr1v=itr2v`. This imposes a restriction on the input program's analysis. Therefore, PolyCheck cannot be employed when aliasing of arrays might occur.

Parallelization As described, PolyCheck assumes a sequential input program and verifies transformations that maintain dependences in this sequential order. However, if the transformed code is a parallel data-race-free program, the checker augmented program is also data-race-free and can be executed in parallel. Specifically, where the transformed program accesses an array location, the checker accesses the location's shadow. Verifying a parallel program then translates to verifying the program meets PolyCheck's check and is data-race free. Note that data-race freedom might be schedule-specific [23, 45] or schedule-independent [16, 44]. PolyCheck's verification is only valid for the schedules shown to be data-race free.

Data space transformations PolyCheck relies on the fact that iteration reordering transformations still maintain the operation order with respect to the data space. Data space transformations that change the array index expressions (e.g., row/column-major to blocked or recursive storage) can lead to valid corresponding statement instances being flagged as in error. However, data space transformations that do not alter the array index expressions (e.g., padding) do not interfere with the checker.

Scalar transformations The current design of PolyCheck cannot handle scalar and basic-block-level transformations. However, some transformations are more difficult than others. Transformations such as common sub-expression elimination and register tiling potentially can be handled by tracking the operation tree representing each assignment. The operation tree represents the set of expressions on array locations in the `InputSet` and `WriteSet` that produce the current value. When a value is assigned to an element in the `WriteSet`, the entire operation tree is checked. Conversely, some transformations, such as constant propagation, change the operations sufficiently to interfere with the construction of the bijection just by tracking their data space effects.

6. Experimental Evaluation

Implementation details PolyCheck first analyzes an input affine program to compute the check codes. This stage was implemented using ISL-0.12 [4] (with barvinok-0.36 and pet-0.04), an integer set library for the polyhedral model, and using the LLVM/Clang-3.4 compiler infrastructure [5]. ISL [51] performs dependence analysis of the source program to generate dependency relations, and the algorithms described in previous sections are implemented to calculate the correct checker for each dependency relation. Then PolyCheck inserts the checker codes for each target statement by analyzing the transformed program to check then performs code generation and outputs the checker-inserted program to enable equivalence checking.

6.1 Evaluation Using the PoCC Polyhedral Compiler

The following is a detailed evaluation of PolyCheck, discussing the coverage and checking of several compiler optimization passes from the PoCC polyhedral research compiler [7].

Benchmarks PolyCheck currently requires the input program to have a static control-flow and use only affine expressions of surrounding loop iterators and program parameters in the loop bounds and array expressions. The PolyBench/C test suite gathers a collection of programs meeting these requirements [8]. Table 5 displays the various computation codes that were evaluated. We used the large dataset size provided.

Table 5: Benchmarks in evaluation using PoCC

Benchmark	Description
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
lu	LU decomposition
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
correlation	Correlation matrix Computation
covariance	Covariance matrix Computation
jacobi-2d-imper	2D Jacobi stencil computation
seidel-2d	2D Seidel stencil computation
fdtd-2d	2D Finite Different Time Domain Kernel
reg_detect	Edge detection
doitgen	Multiresolution analysis kernel (MADNESS)

Tools and setup All benchmark transformations used in evaluation are performed by PoCC-1.2 [7]. To perform comparison experiments with state of the art, we consider the integer set analysis framework (ISA-0.13) [3], a static equivalence checking tool for affine programs using a widening approach.

All running time experiments are performed on Intel Xeon E5-2650 processors at frequency 2.60 GHz with 32 KB L1 cache. The programs are all compiled with GCC-4.8.1 compiler with -O3 optimization. The reported running time is the average of five runs.

6.1.1 Compiler Passes Considered

A polyhedral compiler contains numerous passes, including extracting the polyhedral representation (scop extraction), performing array data-flow analysis (dependence computation), finding an optimized schedule for the program operations (scheduling), and implementing the new schedule as a new C program (code generation). Each of these stages involves several external libraries and possibly tens of thousands of lines of codes. In addition, for better performance numerous complementary AST-based transformations may be performed after code generation, such as unrolling/register-tiling, loop bound optimization, or full tile separation. PoCC [7] implements each of these, and we selected several critical pass combinations to demonstrate our approach.

passthru This configuration performs only SCoP extraction (using the Clan [1] pass), converts the program to its internal polyhedral representation (ScopLib), does not perform any optimization, and generates back a loop-based code implementing the original schedule (using the CLooG [13] pass) that is converted to PoCC's AST representation (PAST) and pretty-printed to a C program. The generated code is an affine program.

data locality This configuration includes all stages of passthru, adding polyhedral data dependence analysis (using the Candl [21] pass) and computation of an optimized schedule of operations to improve temporal data locality and coarse-grain parallelization (using the Pluto algorithm [20]). The generated code is an affine program.

fixed tiling This configuration includes all stages of data locality, adding a modification of the polyhedral representation of statements to implement iteration tiling (a.k.a. loop blocking) whenever possible (using the Pluto [20] pass). Tile sizes are constants provided to the compiler, and the code generation tool attempts to optimize the code structure using the tile size information. The generated code is an affine program.

AST-based unrolling This configuration includes all stages of fixed tiling and unrolls all innermost loops by four, generating an epilog loop to cope with parametric loop bounds if they are not a perfect multiplier of the unroll factor. It stresses the PAST optimizers in PoCC—the AST-based backend optimizers. While the generated code is semantically an affine program, the generated code structure can challenge SCoP extractors in recovering the entire program as a single affine region.

AST-based bound optimization This configuration includes all stages of fixed tiling and performs aggressive loop bound hoisting and simplification, replacing the cascading min/max expressions generated by CLooG for conjunctions of inequalities by a tree of evaluation [56]. While the generated code is still semantically an affine program, recovering the loop bounds expressions roughly requires reverse engineering the optimizations in the SCoP extractor.

parametric tiling This configuration includes all stages of data locality, adding the generation of iteration tiling using parametric tile sizes (using the Ptile [12] pass). That is, the generated code supports arbitrary tile sizes and selecting the tile size at runtime. Because non-affine expressions involving iterators multiplied with parameters (tile sizes) are needed, the generated code is not an affine program,

full tile separation This configuration includes all stages of parametric tiling, adding an AST-based post-processing to separate partial tiles from full tiles in the code structure (using the Ptile [12] pass). The generated code is not an affine program.

6.1.2 Results Overview

Table 6 compares the coverage of PolyCheck and ISA [54], the state-of-the-art equivalence checker tool for affine programs. Logically ISA cannot handle transformed programs that are not affine, such as the parametric tiling cases. ISA is not applicable (N/A) for those. ISA uses the Polyhedral Extraction Tool (PET) to automatically detect affine regions, implementing a powerful SCoP extractor on top of Clang. However, both the AST-based transformations from PoCC evaluated were not handled by the current implementation of the tool (reported as "not supported", N/S). While we believe it is possible to improve the implementation of ISA's SCoP extractor to handle these cases as the generated code still is semantically an affine program, this behavior reinforces the need for a verification tool that is agnostic to how the code has been transformed and generated. PolyCheck fulfills this role and for all optimizations tested can successfully verify equivalence (for correct optimizations) and find bugs (for bugs randomly inserted in the generated code).

Table 6: Summary of tool coverage (PoCC evaluation)

Optimization	PolyCheck	ISA 0.13
passthru	√	√
data locality	√	√
fixed tiling	√	√
AST-based unrolling	√	N/S
AST-based bound optimization	√	N/S
parametric tiling	√	N/A
full tile separation	√	N/A

We have verified the optimizations applied using PoCC for all of the benchmarks listed in Table 5. For all cases where ISA could be applied, both PolyCheck and ISA provide the same positive answer regarding the equivalence between the original and transformed codes. For all other cases where only PolyCheck could be applied, it also reported equivalence between the original and transformed codes. In other words, we have shown the absence of iteration reordering bugs for these test cases in the generated codes.

6.1.3 Finding Bugs in the Generated Codes

To further demonstrate the power of our approach, we emulated bugs in the software by randomly introducing problems in the transformed code (shown in Table 7). Often, bugs translate into an immediate effect of memory corruption (segmentation fault), which usually has visible effects to the user. PoCC already implements a checker that encapsulates all memory references in a wrapper function to detect out-of-bound memory accesses. Here, we assume such a test has already passed. We designed pernicious bugs that do not lead to segmentation fault and that could be easily missed by classical testing procedures that check the bit-by-bit equivalence of the produced outputs by the reference and transformed versions. Indeed, in this case, an open problem is to find datasets that will trigger visible differences in the produced output. In contrast, our approach does not require any reasoning on the input data because we track the satisfaction of data dependences and not the result of the computation performed.

Table 7: Summary of bugs tested (PoCC evaluation)

Bug	Description
loop bound	decrease by 1 some loop upper bound
array access	divide by 2 some array subscript
permutation	interchange two non-permutable loops
code motion	move around some loop nest

A bug has been introduced randomly in each transformed program, for all benchmarks, for both loop bounds and array accesses. For the permutation and code motion, we manually modified the transformed program to introduce a change of semantics when applicable. For some benchmarks, all loop permutations are valid (e.g., dgemm), so no loop permutation bug can be created.

6.1.4 Discussions

Verifying polyhedral transformations Loop transformations, especially iteration reordering ones, are not exclusive to polyhedral compilers. Most compilers implementing some loop transformations will support loop unrolling, interchange, fusion, distribution, and tiling (when possible). When the generated code is a purely affine program, both PolyCheck and ISA can successfully determine the correctness of the transformed program, given the input one. However, the execution time of the two techniques can differ vastly. For ISA, it can prove equivalence extremely quickly, especially on simple programs—in much less than a second. On the other hand, it is at the mercy of polyhedral analysis complexity, which is NP-complete in general. That is, some particular codes could end up taking hours or GB of space to check [54]. In contrast, our approach has a sort of predictable time to terminate: on the order of the time needed for the transformed code to run on a machine, irrespective of the transformation complexity.[3] Still, this time depends on the problem size used.

By definition, polyhedral programs considered have a static control-flow, so it is not necessary to test various datasets of the same size to conclude equivalence with PolyCheck for the scope of bugs it can find. This is in contrast to classical testing, where the output produced by the transformed code is compared bit by bit to the output of a gold, original version. In this case, even for polyhedral programs, the dataset has a fundamental impact on the output values (think about testing dgemm by multiplying 0-filled matrices). We claim PolyCheck is suited to accelerate and even replace such testing procedures because of its insensitivity to dataset values.

Verifying polyhedral and AST transformations It is widely accepted that affine iteration reordering transformations alone are not enough to achieve the best performance. That is, practical polyhedral compilers must complement affine transformations with AST-based transformations for best performance. Shirako et al. recently showed an example of such integration in PoCC [48]. Our approach is robust to complementary transformations implemented at the AST level, as shown with the unrolling experiments. One interesting observation was made in regard to ISA not being able to handle the unrolled code. While the code is still semantically affine, the way it was syntactically generated challenged the ISA SCoP extractor. Clearly, this is not indicative of a limitation of the ISA algorithm/method, merely one of a tool to extract polyhedral representation. Nevertheless, it shows the advantage in having a framework that operates independent of how the code is actually generated. Notably, PolyCheck will not require any update or change for any new optimization applied on affine codes.

[3] Assuming there is no infinite loop in the generated program.

Verifying non-affine transformations Non-affine transformations of affine programs must be properly anticipated for. Existing optimizations, such as parametric tiling, already generate non-affine code and cannot be checked by tools such as ISA. PolyCheck seamlessly handles such codes. The optimization on loop bounds has proven to be another challenge for SCoP extraction in PET. However, this time, the optimization's complexity makes it nearly impossible to create a canonical affine representation of the code at SCoP extraction time without actually reverse-engineering the optimization. To that extent, it can almost be considered as a non-affine transformation, yet there is a compelling need to verify its correctness. Going further, many other code transformations, such as recursive decomposition for cache-oblivious algorithms, lead to non-affine programs, even if the input is an affine program. We believe PolyCheck perfectly complements tools like ISA by enabling seamless support of such transformations, and it offers a more practical alternative to trace-based checking (due to its much lower space complexity) or output difference checking (based on its insensitivity to the dataset content).

6.2 Evaluation Using Cilk and the Pochoir Stencil Compiler

We further evaluate PolyCheck by verifying the correctness of Cilk and Pochoir implementations of affine programs.

Benchmarks and Setup Cilk [18, 19] is a programming model that supports fork/join parallelism based on random work stealing [17]. Cilk programs recursively divide (fork) the computation into sub-computations and combine their result (join) to produce the final result. Cilk is a simple extension of the base C/C++ language with the serial elision property: removing the Cilk keywords results in a sequential recursive program. We evaluated the correctness of the single-threaded execution of the recursive Cilk implementations of affine programs, distributed as part of MIT Cilk 5.4.6 [6]. For each benchmark, we wrote the loop version as the input program specification to verify the Cilk implementation. Then, the checker code was inserted into the Cilk implementation at each statement that generates an operation to be checked.

Pochoir [49, 50] is a embedded domain-specific language for stencil computations. The programmer specifies the computation in terms of a grid and the statements to compute the value of a grid point in a multidimensional spatial grid at time t as a function of neighboring grid points at times before t. Such a specification is compiled into C++ by the Pochoir compiler to generate divide-and-conquer stencil codes [24, 25] based on cache-oblivious algorithms [26, 40]. Note that the iterations or loop space are completely implicit in the program and immediately not available to the programmer. Each benchmark in the Pochoir distribution (version 0.5) includes a reference loop implementation used for testing. We engaged these implementations as the input source program. The computation to be performed at each grid point is a set of statements. We inserted a checker code to verify each instance of these statements directly in the Pochoir program. The program is then compiled through Pochoir and executed to verify the Pochoir compiler's transformations.

Table 9 shows the Cilk and Pochoir benchmarks used in the evaluation. The default dataset size was used for all benchmarks. All benchmarks were compiled with the ICC-15.0.3 compiler with -O3 optimization.

Evaluation and results The evaluation of our approach on Cilk and Pochoir programs is similar to Polybench/C test suite. To emulate bugs, we randomly introduce problems to programs (shown in Table 10). As in the PoCC study, we ensure the bugs introduced do not lead to segmentation violation.

All bugs are introduced randomly for all transformed programs, loop bound, domain, and array access. Of note, there are no explicit

Table 8: Errors found in transformations by PolyOpt/C 0.2.0

Benchmark	Original Program		Transformed Program with tile_8_1_1(involves errors)	
cholesky	`A[1][0](1)=x(2)*p[0](1)` `assert( x(2) == 2 )`	PASS	`A[1][0](1)=x(1)*p[0](1)` `assert( x(1) == 2 )`	Failure
reg_detect	`mean[0][0](1)=sum_diff[0][0][15](1)` `assert( sum_diff[0][0][15](1) == 1)`	PASS	`mean[0][0](1)=sum_diff[0][0][15](0)` `assert( sum_diff[0][0][15](0) == 0 )`	Failure

Table 9: Cilk and Pochoir benchmarks

Benchmarks		Description
Cilk	matmul	Matrix-multiply C=A.B
	rectmul	Multiply two rectangualar matrizes
	spacemul	A dag-consistent Matrix Multiply
	heat	Heat diffusion
	lu	Martix LU decomposition
Pochoir	heat_2D	heat equation on 2D grid
	heat_2P	heat equation on 2D torus
	apop	American put stock option pricing
	3d7pt	order-1 3D 7 point stencil
	3d27pt	order-1 3D 27-point stencil

Table 10: Summary of evaluation using Cilk and Pochoir benchmarks

Evaluation		Description	Detect
Cilk	no bug	original Cilk program	Pass
	loop bound	decrease some loop upper bound	$\surd$
	array access	decrease some array subscript	$\surd$
Pochoir	no bug	original Pochoir program	Pass
	loop domain	decrease some loop domain	$\surd$
	array access	decrease some array subscript	$\surd$

loops in the Pochoir program, but macro functions specify the loop domain. Table 10 shows the results.

6.3 Identifying Bugs in PolyOpt/C

CodeThorn [46] identified two bugs in PolyOpt/C 0.2.0 [39] polyhedral compiler. These bug results in incorrect code generated for the cholesky and reg_detect benchmarks. CodeThorn checks the transformed program for a given problem size by explicitly enumerating the trace of statement instances and performing a verification. Table 8 shows these bugs, illustrating the array element following the version number (in parenthesis). PolyCheck found both bugs efficiently. We also present performance comparisons with CodeThorn below, which showcase how PolyCheck can be orders of magnitude faster than CodeThorn.

6.4 PolyCheck Overhead

Runtime overhead We conclude our evaluation with a detailed reporting of the execution time of the transformed programs modified to integrate the checkers. To evaluate the checker-only overhead, we replaced the actual computation with the checker's actions. Each program must be run to completion to provide verification information. Figure 9 reports the timing (normalized to the execution time of the transformed program) of the fixed-tiling transformation, without and with checker optimization described earlier. Figure 10 shows the same but for the parametric-tiling transformation. We remark that for reg_detect, the checker code is initially slower than the transformed code. This is because our checker code can disrupt SIMD vectorization optimizations, an effect exacerbated for reg_detect. In general, the checker's execution time is proportional to the performance of the transformed program because it has almost identical memory traffic. The arithmetic operations performed in the actual computation are replaced by checker instructions, which can possibly represent more workload for the checker version. The checker optimization dramatically reduces any such effect (shown in both Figures 9 and 10). The -Opt timing is systematically lower than the transformed program, and there are cases, such as bicg and lu, where the overhead becomes marginal. Nevertheless, as we always execute the program, the execution time remains dependent on the problem size. In Figure 11, we show the execution time to check fixed- and parametric-tiled versions of LU. It is evident that the time to execute the checkers is significantly lower than the transformed program. This is due to the use of optimized checking of full tiles. Figure 12 shows the execution time for reg_detect, where full tile optimization cannot applied.

Table 11: Overhead comparison with CodeThorn [46]

Benchmark	CodeThorn	PolyCheck
covariance	1.56 secs	0.005 ms
covariance-tile-8-1-1	1.71 secs	0.030 ms
fdtd-2d	1.58 secs	0.047 ms
fdtd-2d-tile-8-1-1	3.14 secs	0.071 ms
jacobi-2d-imper	0.692 ms	0.027 ms
jacobi-2d-imper-tile-8-1-1	1.50 secs	0.037 ms
seidel-2d	1.05 secs	0.031 ms
seidel-2d-tile-8-1-1	2.51 secs	0.032 ms

Overhead Comparison We further evaluate PolyCheck's runtime overhead by presenting a comparison experiment of CodeThorn, a trace-based tool by Schordan et al. [46]. Table 11 compares the two tools using benchmarks and problem sizes chosen from [46]. CodeThorn runtimes are directly from Schordan et al. [46], and we report the time of the optimized runtime checking procedure only for PolyCheck (the static analysis time does not depend on the problem size and needs to be done only once per *input* program). PolyCheck can be orders of magnitude faster than codeThorn, thanks to very efficient runtime checking by actual execution of the program, and limited space overhead. However PolyCheck requires the input program to have a static/affine control-flow, while codeThorn can handle programs with data-dependent control-flow.

7. Related Work

Verdoolaege et al. proposed a fully automatic technique to prove equivalence between two affine programs [54]. Leveraging polyhedral data-flow analysis, they developed widening/narrowing operators to properly handle non-uniform recurrences. It is implemented in the ISA tool [3]. However, in contrast to our approach, it is limited to verifying affine program transformations. Basupalli et al. developed ompVerify, to find OpenMP parallelization errors in affine programs [14]; and Alias et al. [9, 11] have developed other techniques to recognize algorithm templates in programs. These approaches are restricted to static/affine transformed programs. Karfa et al. also designed a method that works for a subset of affine

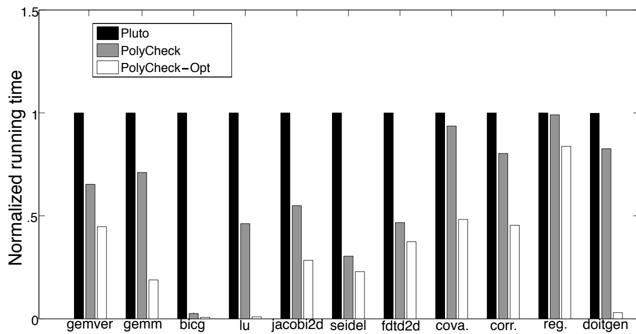

Figure 9: Checker running time with fixed tiling.

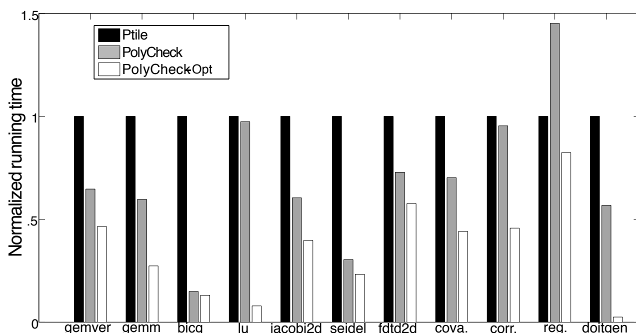

Figure 10: Checker running time with parametric tiling.

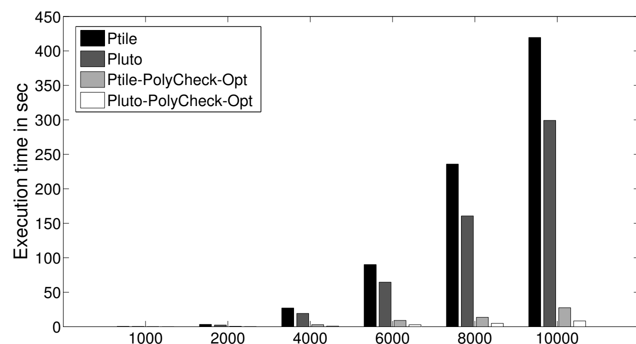

Figure 11: Checker running time across problem size (LU).

programs using array data dependence graphs (ADDGs) to represent input and transforming behaviors. Operator-level equivalence checking provides the capability to normalize expressions and establish matching relations under algebraic transformations [33].

Recently, Schordan et al. proposed a trace-based framework to verify if two programs (one possibly being a transformed variant of the other) are semantically equivalent. Their method combines the computation of a state transition graph with a rewrite system to transform floating point computations and array update operations of one program to match them as terms with those of the other program [46]. In contrast to our approach, which only requires the same space as the input program's working dataset size, the space complexity in their approach is a function of the total number of dynamic instances of operations.

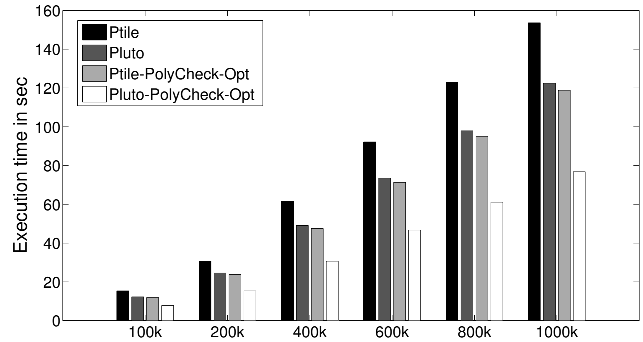

Figure 12: Checker running time across problem size (Reg_detect).

Other related works include Mansky and Gunter, who used the TRANS language [32] to represent transformations. The correctness proof implemented in the verification framework [35] is verified by the Isabelle [38] proof assistant. Regression verification has been considered to support recursion, but it actually requires loops to be converted to recursion first [29]. Other works also include translation validation [34, 37].

Extending or simplifying static analysis through dynamic analysis of the program's execution has been considered in prior work [15, 31, 36, 47]. In the future, we plan to investigate analogous extensions to our dynamic verification approach to reason about program equivalence in a problem-size-independent manner.

8. Conclusions

Using compositions of loop transformations to restructure the program for improved performance, optimizing compilers become increasingly complex and capable of generating transformed programs that are extremely far from the original code syntactically. It is critical to assess the correctness of compiler-generated code as these compilation frameworks themselves rely on millions of lines of code. In this paper, we presented a new approach that exploits the properties of affine programs to generate, at compile time, a lightweight checking code. This checking code then is embedded into the transformed program and run for equivalence checking. Our approach addresses the main drawbacks of alternative solutions for finding bugs in iteration reordering transformation frameworks, and its correctness and effectiveness have been extensively evaluated on several compilation frameworks that combine numerous kinds of program transformations.

Acknowledgments

We thank the anonymous referees for the feedback and many suggestions that helped us significantly in improving the presentation of the work. This work was supported in part by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under awards 63823, 66905, and DE-SC0014135, and the U.S. National Science Foundation through award 1321147. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] Clan, the Chunky Loop Analyzer. `http://icps.u-strasbg.fr/~bastoul`.

[2] GNU GCC. `http://gcc.gnu.org`.

[3] ISA 0.13. `http://repo.or.cz/w/isa.git`.

[4] ISL, the Integer Set Library. `http://repo.or.cz/w/isl.git`.

[5] LLVM. http://llvm.org.

[6] MIT Cilk. http://supertech.csail.mit.edu/cilk.

[7] PoCC, the Polyhedral Compiler Collection 1.3. http://pocc.sourceforge.net.

[8] PolyBench/C 3.2. http://polybench.sourceforge.net.

[9] C. Alias and D. Barthou. On the recognition of algorithm templates. *Electronic Notes in Theoretical Computer Science*, 82(2):395–409, 2004.

[10] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan. Polycheck: Dynamic verification of iteration space transformations on affine programs. Technical report, OSU/PNNL/INRIA, Nov. 2015. OSU-CISRC-11/15-TR21.

[11] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Euro-Par 2002 Parallel Processing*. 2002.

[12] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proc. of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010.

[13] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2004.

[14] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompVerify: polyhedral analysis for the OpenMP programmer. In *OpenMP in the Petascale Era*, pages 37–53. Springer, 2011.

[15] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur. Proofs from tests. In *Proc. of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. IEEE, 2010.

[16] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proc. of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*. ACM, 2004.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 1995.

[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[20] 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*. ACM, 2008.

[21] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[22] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[23] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, 2009.

[24] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proc. of the 19th annual international conference on Supercomputing*. ACM, 2005.

[25] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[26] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science*. IEEE, 1999.

[27] P. Gachet, C. Mauras, P. Quinton, and Y. Saouter. Alpha du centaur: a prototype environment for the design of parallel regular alorithms. In *Proc. of the 3rd international conference on Supercomputing*. ACM, 1989.

[28] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[29] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.

[30] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6):607–631, 2000.

[31] A. K. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *Proc. of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*. Springer, 2009.

[32] S. Kalvala, R. Warburton, and D. Lacey. Program transformations using temporal logic side conditions. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 31(4):14, 2009.

[33] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32(11):1787–1800, 2013.

[34] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. *ACM SIGPLAN Notices*, 44(6):327–337, 2009.

[35] W. Mansky and E. Gunter. A framework for formal verification of compiler optimizations. In *Interactive Theorem Proving*. Springer, 2010.

[36] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *Proc. of the 39th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'12)*. ACM, 2012.

[37] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[38] L. C. Paulson. Isabelle Page. https://www.cl.cam.ac.uk/research/hvg/Isabelle.

[39] L. Pouchet. Polyopt/C: A polyhedral optimizer for the rose compiler, 2011.

[40] H. Prokop. *Cache-oblivious algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.

[41] D. Quinlan, C. Liao, R. Matzke, M. Schordan, T. Panas, R. Vuduc, and Q. Yi. ROSE Web Page. http://www.rosecompiler.org, 2014.

[42] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):95–113, 1989.

[43] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proc. of the 16th annual conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 1986.

[44] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proc. of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, 2012.

[45] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[46] M. Schordan, P.-H. Lin, D. Quinlan, and L.-N. Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Proc. of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2014.

[47] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Proc. of the 22nd European conference on Programming Languages and Systems (ESOP'13)*. Springer, 2013.

[48] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and water can mix: Reconciling polyhedral and ast transformations. In *IEEE/ACM Conference on Supercomputing (SC'14)*. IEEE, 2014.

[49] Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson. Coding stencil computations using the pochoir stencil-specification language. In *Poster session presented at the 3rd USENIX Workshop on Hot Topics in Parallelism*, 2011.

[50] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proc. of the 32rd annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011.

[51] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *The 3rd International Congress on Mathematical Software (ICMS'10)*. Springer, 2010.

[52] S. Verdoolaege. Counting affine calculator and applications. In *The 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, 2011.

[53] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[54] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 34 (3):11, 2012.

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

[56] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong. Improving polyhedral code generation for high-level synthesis. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*. IEEE, 2013.