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

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,
We present the first dynamic bug checker for affine programs 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 CodeThorn’s trace-based equivalence checker. 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 static equivalence checking schemes. Finally, 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.

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 the transformed 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., $\sum$) 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 statement instance associated...
The program is executed. The checker constructed for this particular instance is as follows: the 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.

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 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 interchangeably.

---

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:

\[
S \equiv [p_1, ..., p_m] \rightarrow \{(i_1, ..., i_m) : c_1 \land \ldots \land c_n\}
\]

where \( i_1, ..., i_m \) index the \( m \) dimensions of the set (noted \( \vec{i} \)); \( p_1, ..., p_m \) 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 \( IS \) with one component per surrounding loop, and the values \( r_s \) can take are captured by defining its iteration space \( IS \). The iteration space of the statement \( S_1 \) in Figure 2 is noted \( IS_1 \), and is:

\[
IS_1 = [T, N] \rightarrow \{S_1[t, i, j] : 0 \leq t < T \land 0 \leq i < N \land 0 \leq j < N\}
\]

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

\[
r = [p_1, ..., p_m] \rightarrow \{(i_1, ..., i_m) \mapsto (j_1, ..., j_m) : c_1 \land \ldots \land c_n\}
\]

Relations are used to describe the set of memory locations accessed by statements. In polyhedral programs, array subscripts 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:

\[
R_S^{A_1} = \{S_1[t, i, j] \Rightarrow A[i_1, j_1] : (i_1 = i) \land (j_1 = j)\}
\]

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

**Applying relations to sets** The apply operation is defined as:

\[
(\vec{x} \in s') \iff (\exists \vec{y} : \vec{x} \in r \Rightarrow \vec{y} \in s')
\]

where \( s' \) is a new set produced by applying \( r \) 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 \( W_S^{A_1}(IS_1) \).

**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 \( (a_1, ..., a_n) \prec (b_1, ..., b_m) \) iff there exists an integer \( 1 \leq i \leq \min(n, m) \) st. \( a_1, ..., a_{i-1} = b_1, ..., b_{i-1} \) 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:

\[
\text{Sched}_S = \{S_1[t, i, j] \rightarrow [0, t, 0, i, 0, j, 0]\}
\]

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:

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

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:

```latex
/*@ polycheck start spec
<braffine input program>
*/
<br 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 \( op \) attempts to write to memory location \( l \). The last valid iteration \( S<itrv>^2 \) that wrote \( l \) is stored in \text{ashadow}(l). From there, we

---

2 Throughout the paper, we use multi-character identifiers that end in \( v, \) e.g. \text{itrv}, to denote vectors.
can determine what should be $S<itr2v>$, the next iteration writing to $i$, by evaluating a function we built via static analysis 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 $S<itr2v>$ was the producing iteration for each memory location accessed by $S<itr2v>$. This information is then checked against the runtime information of $o$: 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 $o$ is an invalid operation, and the transformed program does not match the input program. If all operations $o$ are valid and every $o$ was matched with exactly one iteration $S<itr2v>$ from the input program, the transformed program 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 transformed code. These functions use various integer set/map operations, such as union ($\cup$), computing the lexicographic minimum ($\text{lexmin}$) and maximum ($\text{lexmax}$), and application of a map to a set ($M(S)$).

**FirstWriter** The function $\vec{i} = \text{FirstWriter}(A, \vec{w})$ computes which instance $\vec{i}$ 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 parameter point in a set:

$$\text{Point}_A = [\vec{W}] \rightarrow \{[\vec{w}] : \vec{w} = \vec{W}\}$$

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:

$$M_A^S = [\vec{P}] \rightarrow \{[\vec{w}] \rightarrow [\vec{i}] : W_A^S(\vec{i}) = \vec{w} \land \vec{i} \in \mathcal{I}_S\}$$

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:

$$T = \text{Sched}(M_A^S(\text{Point}_A))$$

Finally, $S$ 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($A$, $\vec{w}$) =

$$\text{lexmin} \left( \bigcup_{S \in S} \text{Sched}(M_A^S(\text{Point}_A)) \right)$$

We remark that the preceding sets and relations, including the $\text{lexmin}$ operation, can be seamlessly computed using, for instance, the ISCC calculator. The $\text{lexmin}$ returned is an expression (typically a tree of expressions, where leaves are possible $\text{lexmin}$ values and nodes in the tree are conditions on the numerical values of $\vec{w}$ and $\vec{P}$). To build the function FirstWriter($A$, $\vec{w}$), 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 $\text{lexmin}$ expression tree to evaluate as a function of the array name considered.

**NextWriter** The function $\vec{i} = \text{NextWriter}(\vec{f}_{\text{prev}}, A, \vec{w})$ computes which is the instance $\vec{i}$ in the input program writing to the location $\vec{w}$ of $A$ executing immediately after $f_{\text{prev}}$ 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:

$$\text{Iter1}(\vec{T}) = \{[\vec{i}] : \vec{i} = \vec{T}\}$$

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 $\vec{i}$ immediately following another iteration $\vec{f}_{\text{prev}}$, we use a slight extension of the definition of $M_A^S$ to add input dimensions and capture $f_{\text{prev}}$ as follows:

$$N_A^S = [\vec{P}] \rightarrow \{[\vec{w}, \vec{f}_{\text{prev}}] \rightarrow [\vec{i}] : W_A^S(\vec{i}) = \vec{w} \land \vec{i} \in \mathcal{I}_S \land f_{\text{prev}} \prec \text{Sched}(\vec{i})\}$$

Finally NextWriter($\vec{f}_{\text{prev}}, A, \vec{w}$) is built using a similar expression as for FirstWriter($A$, $\vec{W}$), simply substituting $M_A^S$ by $N_A^S$ and Point$_A$ by (Iter1, Point$_A$).

**WriterBeforeRead** $\vec{i} = \text{WriterBeforeRead}(\vec{f}_{\text{read}}, A, \vec{w})$ computes the instance $\vec{i}$ that last wrote to a location $\vec{w}$ of array $A$ read by iteration $\vec{f}_{\text{read}}$ 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 $f_{\text{prev}} \prec \text{Sched}(\vec{i})$ we state $\text{Sched}(\vec{i}) \prec \vec{f}_{\text{read}}$, and compute the lexmax of the problem instead of the lexmin to have the instance immediately preceding $f_{\text{read}}$ 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 $\vec{i}$ is set to $\text{Init<0>}$.

**LastWriter** The function $\vec{i} = \text{LastWriter}(A, \vec{w})$ computes the instance $\vec{i}$ 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 $\vec{i}$.

**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:

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

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 $\text{Init<0>}$. It is assembled by first building the set of all memory locations being read. For the array $A$, it is:

$$\text{ReadSet}(A) = \bigcup_{S \in S} \left( R_A^S(\mathcal{I}_S) \right),$$

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 $\text{Init<0>}$. 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 \( w^* \) in \( A \) is \( A[w^*] \).

**Notations** An array location is denoted by the array label and index vector (e.g., \( A[indexv] \)), where \( indexv \) 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., \( S<itrvec> \)).

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 \( S<itrvec> \) is computed as \( S.w(itrvec) \).

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 \( S<itrvec> \) is computed as \( S.r[0](itrvec), S.r[1](itrvec), etc. \)

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

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 program 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.

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/maximun 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.

```plaintext
initialize:
// executed before starting execution of transformed program
for w in WriteSet:
    shadow(w) = NIL
for i in InputSet:
    shadow(i) = Init<0>

def input_statement_instance(o):
    assert shadow(o.Aw[o.wv])
    if shadow(o.Aw[o.wv]) == NIL:
        S<itrvec> = FirstWriter(o.Aw[o.wv])
    else:
        S<itrvec> = NextWriter(shadow(o.Aw[o.xv]),
                                o.Aw[o.wv])
    assert S<itrvec> exists //valid iteration
    assert S in CandidateInputStmts(ts(o))
    return S<itrvec>

@verify(o):
// executed for every operation encountered in the transformed program
S<itrvec> = input_statement_instance(o)
for i in 0..|S.r[1]|:
    assert S.r[1][itrvec] == o.rv[i]
    assert RAW(S,i)(itrvec) == shadow(o.Ar[i][o.rv[i]])
    shadow(o.Aw[o.wv]) = S<itrvec>
@terminate:
// executed after execution the transformed program
for w in WriteSet:
    assert shadow(w) == LastWriter(w)
report SUCCESS
```

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

**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., \( Ts \)).

A VersionNumber \( S \) function is built such that, given a instance \( S<itrvec> \) 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 technique [53] available in ISCC. The function RAWv(S,i) returns the VersionNumber of RAW(S,i).

The function NumWriters(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
Proof. Proof is presented in [10].

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.

The right-hand side of the equation to solve for \( x \) in \( \text{LIN}(S).xv = \text{RHS}(S) \).

**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.

```python
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 \( \text{LIN}(S) \). xv = \( \text{RHS}(S)(o.w, o.rv[i], ..) \) s.t. xv in LB(S)
12        if xv is found:
13           // check below is redundant if
14           // VersionNumber(S) is a linear function
15           if VersionNumber(S)(xv) == shadow(o.Aw[o.wv]):
16             return S<xv>
17             assert false // does not match any input statement instance
18
19 @verify(o):
20 //executed for every operation encountered in the transformed program
21 S<itr> = input_statement_instance(o)
22 for i in 0 .. |S.r|-1:
23    if RAMv(S,i) not in LIN(S): // non-linear component
24        assert shadow(o.Ar[i][o.rv[i]]) exists
25        assert RAMv(S,i)({itr} == shadow(o.Ar[i][o.rv[i]])
26    shadow(o.Aw[o.wv]) += 1
27
28 @terminate:
29 //executed after execution of the transformed program
30 for w in WriteSet:
31     assert shadow(w) == NumWrites(w)
32 report SUCCESS
```

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 1: FirstWriter to array A in iterative Seidel](image)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>S[0,1]</td>
<td>S[1,1]</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>S[0,2]</td>
<td>S[2,2]</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>S[0,3]</td>
<td>S[3,3]</td>
</tr>
</tbody>
</table>

![Figure 5: Runtime procedure to check every operation encountered in the transformed program for algorithm B.](image)

![Figure 6: Recursive implementation of Seidel. Invoked as seidel_rec(0,1,3,1,3,2,4,A).](image)
Table 2: Statement instances executed by iterative Seidel and related relations

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Write</th>
<th>NextW</th>
<th>Read1</th>
<th>PrevW</th>
<th>Read2</th>
<th>PrevW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&lt;0,1.1&gt;</td>
<td>A[1,1]</td>
<td>S&lt;1,1.1&gt;</td>
<td>A[0,1]</td>
<td>A[0,1]</td>
<td>A[1,0]</td>
<td></td>
</tr>
<tr>
<td>S&lt;0,1.2&gt;</td>
<td>A[1,2]</td>
<td>S&lt;1,1.2&gt;</td>
<td>A[0,2]</td>
<td>-</td>
<td>A[1,1]</td>
<td>S&lt;0,1.1&gt;</td>
</tr>
<tr>
<td>S&lt;0,1.3&gt;</td>
<td>A[1,3]</td>
<td>S&lt;1,1.3&gt;</td>
<td>A[0,3]</td>
<td>-</td>
<td>A[1,2]</td>
<td>S&lt;0,1.2&gt;</td>
</tr>
<tr>
<td>S&lt;0,2.1&gt;</td>
<td>A[2,1]</td>
<td>S&lt;1,2.1&gt;</td>
<td>A[1,1]</td>
<td>S&lt;0,1.1&gt;</td>
<td>A[2,0]</td>
<td>-</td>
</tr>
<tr>
<td>S&lt;0,2.2&gt;</td>
<td>A[2,2]</td>
<td>S&lt;1,2.2&gt;</td>
<td>A[1,2]</td>
<td>S&lt;0,1.2&gt;</td>
<td>A[2,1]</td>
<td>S&lt;0,2,1&gt;</td>
</tr>
<tr>
<td>S&lt;0,2,3&gt;</td>
<td>A[2,3]</td>
<td>S&lt;1,2,3&gt;</td>
<td>A[1,3]</td>
<td>S&lt;0,1,3&gt;</td>
<td>A[2,2]</td>
<td>S&lt;0,2,2&gt;</td>
</tr>
<tr>
<td>S&lt;0,3,1&gt;</td>
<td>A[3,1]</td>
<td>S&lt;1,3,1&gt;</td>
<td>A[2,1]</td>
<td>S&lt;0,2,1&gt;</td>
<td>A[3,0]</td>
<td>-</td>
</tr>
<tr>
<td>S&lt;1,1,1&gt;</td>
<td>A[1,1]</td>
<td>-</td>
<td>A[0,1]</td>
<td>-</td>
<td>A[1,0]</td>
<td>-</td>
</tr>
<tr>
<td>S&lt;1,1,2&gt;</td>
<td>A[1,2]</td>
<td>-</td>
<td>A[0,2]</td>
<td>-</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
</tr>
<tr>
<td>S&lt;1,1,3&gt;</td>
<td>A[1,3]</td>
<td>-</td>
<td>A[0,3]</td>
<td>-</td>
<td>A[1,2]</td>
<td>S&lt;0,1,2&gt;</td>
</tr>
<tr>
<td>S&lt;1,2,1&gt;</td>
<td>A[2,1]</td>
<td>-</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
<td>A[2,0]</td>
<td>-</td>
</tr>
<tr>
<td>S&lt;1,2,2&gt;</td>
<td>A[2,2]</td>
<td>-</td>
<td>A[1,2]</td>
<td>S&lt;0,1,2&gt;</td>
<td>A[2,1]</td>
<td>S&lt;0,2,1&gt;</td>
</tr>
<tr>
<td>S&lt;1,2,3&gt;</td>
<td>A[2,3]</td>
<td>-</td>
<td>A[1,3]</td>
<td>S&lt;0,1,3&gt;</td>
<td>A[2,2]</td>
<td>S&lt;0,2,2&gt;</td>
</tr>
<tr>
<td>S&lt;1,3,1&gt;</td>
<td>A[3,1]</td>
<td>-</td>
<td>A[2,1]</td>
<td>S&lt;0,2,1&gt;</td>
<td>A[3,0]</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Write</th>
<th>Sdw_old</th>
<th>Sdw_new /Match</th>
<th>Read1</th>
<th>Shadow</th>
<th>Read2</th>
<th>Shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[1,1]</td>
<td>-</td>
<td>S&lt;0,1.1&gt;</td>
<td>A[0,1]</td>
<td>-</td>
<td>A[1,0]</td>
<td>-</td>
</tr>
<tr>
<td>A[1,2]</td>
<td>-</td>
<td>S&lt;0,1,2&gt;</td>
<td>A[0,2]</td>
<td>-</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
</tr>
<tr>
<td>A[2,1]</td>
<td>-</td>
<td>S&lt;0,2,1&gt;</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
<td>A[2,0]</td>
<td>-</td>
</tr>
<tr>
<td>A[2,2]</td>
<td>-</td>
<td>S&lt;0,2,2&gt;</td>
<td>A[1,2]</td>
<td>S&lt;0,1,2&gt;</td>
<td>A[2,1]</td>
<td>S&lt;0,2,1&gt;</td>
</tr>
<tr>
<td>A[3,1]</td>
<td>-</td>
<td>S&lt;0,1,3&gt;</td>
<td>A[0,3]</td>
<td>-</td>
<td>A[1,2]</td>
<td>S&lt;0,1,2&gt;</td>
</tr>
<tr>
<td>A[3,2]</td>
<td>-</td>
<td>S&lt;0,2,3&gt;</td>
<td>A[1,3]</td>
<td>S&lt;0,1,3&gt;</td>
<td>A[2,2]</td>
<td>S&lt;0,2,2&gt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Write</th>
<th>Sdw_old</th>
<th>Sdw_new /Match</th>
<th>Read1</th>
<th>Shadow</th>
<th>Read2</th>
<th>Shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[1,1]</td>
<td>-</td>
<td>S&lt;0,1.1&gt;</td>
<td>A[0,1]</td>
<td>-</td>
<td>A[1,0]</td>
<td>-</td>
</tr>
<tr>
<td>A[1,2]</td>
<td>-</td>
<td>S&lt;0,1,2&gt;</td>
<td>A[0,2]</td>
<td>-</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
</tr>
<tr>
<td>A[2,1]</td>
<td>-</td>
<td>S&lt;0,2,1&gt;</td>
<td>A[1,1]</td>
<td>S&lt;0,1,1&gt;</td>
<td>A[2,0]</td>
<td>-</td>
</tr>
</tbody>
</table>

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.
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:

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

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

```c
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, we place the program 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 accessed during the verification. This imposes a restriction 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’ than 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.

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 dependencies 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 an 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>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>Matrix-multiply C=alpha.A.B+beta.C</td>
</tr>
<tr>
<td>gemver</td>
<td>Vector Multiplication and Matrix Addition</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>bicg</td>
<td>BiCG Sub Kernel of BiCGStab Linear Solver</td>
</tr>
<tr>
<td>correlation</td>
<td>Correlation matrix Computation</td>
</tr>
<tr>
<td>covariance</td>
<td>Covariance matrix Computation</td>
</tr>
<tr>
<td>jacob2d-imper</td>
<td>2D Jacobi stencil computation</td>
</tr>
<tr>
<td>seidel2d</td>
<td>2D Seidel stencil computation</td>
</tr>
<tr>
<td>ffdl2d</td>
<td>2D Finite Different Time Domain Kernel</td>
</tr>
<tr>
<td>reg_detect</td>
<td>Edge detection</td>
</tr>
<tr>
<td>doitgen</td>
<td>Multiresolution analysis kernel (MADNESS)</td>
</tr>
</tbody>
</table>

Table 5: Benchmarks in evaluation using PoCC
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 code. 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 SCOPE 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 AST optimizers in PoCC—the AST-based backend optimizers. While the generated code is semantically an affine program, the generated code structure can challenge SCOPE 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 SCOPE 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 SCOPE 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 SCOPE 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)

<table>
<thead>
<tr>
<th>Optimization</th>
<th>PolyCheck</th>
<th>ISA 0.13</th>
</tr>
</thead>
<tbody>
<tr>
<td>passthru</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>data locality</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>fixed tiling</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>AST-based unrolling</td>
<td>√</td>
<td>N/S</td>
</tr>
<tr>
<td>AST-based bound optimization</td>
<td>√</td>
<td>N/S</td>
</tr>
<tr>
<td>parametric tiling</td>
<td>√</td>
<td>N/A</td>
</tr>
<tr>
<td>full tile separation</td>
<td>√</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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.
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 detect bugs, we randomly introduce problems to programs (shown in Table 7).

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 an 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 PoC 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 bugs introduced for polyhedral programs, the dataset has a fundamental impact on how the code is actually generated. Nevertheless, it shows the advantage in having a framework to extract polyhedral representations. Of note, there is no infinite loop in the generated program.
code can disrupt SIMD vectorization optimizations, an effect exac-
tically slower than the transformed code. This is because our checker
formation. We remark that for
lier. Figure 10 shows the same but for the parametric-tiling trans-
tions. Each program must be run to completion to provide verifica-
head, we replaced the actual computation with the checker's ac-
ified to integrate the checkers. To evaluate the checker-only over-
Runtime overhead

6.4 PolyCheck Overhead

Runtime overhead We conclude our evaluation with a detailed
reporting of the execution time of the transformed programs modi-
tified to integrate the checkers. To evaluate the checker-only over-
head, we replaced the actual computation with the checker’s ac-
ctions. Each program must be run to completion to provide verifica-
tion information. Figure 9 reports the timing (normalized to the ex-
cution time of the transformed program) of the fixed-tiling trans-
formation, without and with checker optimization described ear-
lier. Figure 10 shows the same but for the parametric-tiling trans-
formation. We remark that for reg_detect, the checker code is ini-
tially slower than the transformed code. This is because our checker
code can disrupt SIMD vectorization optimizations, an effect exac-

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original Program</th>
<th>Transformed Program with tile_8_1_1 (involves errors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cholesky</td>
<td>A(1)<a href="1">0</a>=x(2)+p<a href="1">0</a> assert( x(2) == 2 )</td>
<td>A(1)<a href="1">0</a>=x(2)+p<a href="1">0</a> assert( x(1) == 2 )</td>
</tr>
<tr>
<td>reg_detect</td>
<td>mean(0)<a href="1">0</a>=sum_diff[0][0]<a href="1">15</a> assert( sum diff[0][0]<a href="1">15</a> == 1)</td>
<td>mean(0)<a href="1">0</a>=sum_diff[0][0]<a href="0">15</a> assert( sum diff[0][0]<a href="0">15</a> == 0 )</td>
</tr>
</tbody>
</table>

Table 9: Cilk and Pochoir benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>mmat - Matrix-multiply C=A*B</td>
</tr>
<tr>
<td></td>
<td>rectmat - Multiply two rectangular matrices</td>
</tr>
<tr>
<td></td>
<td>spacemat - A dag-consistent Matrix Multiply</td>
</tr>
<tr>
<td></td>
<td>heat - Heat diffusion</td>
</tr>
<tr>
<td></td>
<td>lu - Matrix LU decomposition</td>
</tr>
<tr>
<td></td>
<td>heat_2D - heat equation on 2D grid</td>
</tr>
<tr>
<td></td>
<td>heat_2P - heat equation on 2D torus</td>
</tr>
<tr>
<td></td>
<td>aop - American put stock option pricing</td>
</tr>
<tr>
<td></td>
<td>3D/pt - order-1 3D 7 point stencil</td>
</tr>
<tr>
<td></td>
<td>3D/pt - order-1 3D 27-point stencil</td>
</tr>
<tr>
<td>Pochoir</td>
<td>covariance - Heat diffusion</td>
</tr>
<tr>
<td></td>
<td>covar-tile-8-1-1 - Covariance-tiled 8x1x1</td>
</tr>
<tr>
<td></td>
<td>covariance-tile-8-1-1 - Covariance-tiled 8x1x1</td>
</tr>
<tr>
<td></td>
<td>fdtd-2d - Heat diffusion</td>
</tr>
<tr>
<td></td>
<td>fdtd-2d-tile-8-1-1 - Heat diffusion-tiled 8x1x1</td>
</tr>
<tr>
<td></td>
<td>jacob-2d-imper - Jacobian</td>
</tr>
<tr>
<td></td>
<td>jacob-2d-imper-tile-8-1-1 - Jacobian-tiled 8x1x1</td>
</tr>
<tr>
<td></td>
<td>seidel-2d - Seidel</td>
</tr>
<tr>
<td></td>
<td>seidel-2d-tile-8-1-1 - Seidel-tiled 8x1x1</td>
</tr>
</tbody>
</table>

Table 10: Summary of evaluation using Cilk and Pochoir benchmarks

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Description</th>
<th>Detect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>no bug</td>
<td>original Cilk program</td>
</tr>
<tr>
<td></td>
<td>loop bound</td>
<td>decrease some loop upper bound</td>
</tr>
<tr>
<td></td>
<td>array access</td>
<td>decrease some array subscript</td>
</tr>
<tr>
<td>Pochoir</td>
<td>no bug</td>
<td>original Pochoir program</td>
</tr>
<tr>
<td></td>
<td>loop domain</td>
<td>decrease some loop domain</td>
</tr>
<tr>
<td></td>
<td>array access</td>
<td>decrease some array subscript</td>
</tr>
</tbody>
</table>

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] poly-
hedral compiler. These bugs result in incorrect code generated for the
cholesky and reg_detect benchmarks. CodeThorn checks the
transformed program for a given problem size by explicitly enu-
merating the trace of statement instances and performing a verifica-
tion. 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 modi-
tified to integrate the checkers. To evaluate the checker-only over-
head, we replaced the actual computation with the checker’s ac-
ctions. Each program must be run to completion to provide verifica-
tion information. Figure 9 reports the timing (normalized to the ex-
cution time of the transformed program) of the fixed-tiling trans-
formation, without and with checker optimization described ear-
lier. Figure 10 shows the same but for the parametric-tiling trans-
formation. We remark that for reg_detect, the checker code is ini-
tially slower than the transformed code. This is because our checker
code can disrupt SIMD vectorization optimizations, an effect exac-

7. Related Work

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


