Verification of Polyhedral Optimizations with Constant Loop Bounds in Finite State Space Computations

Markus Schordan[1], Pei-Hung Lin[1], Dan Quinlan[1], and Louis-Noël Pouchet[2]

[1] Lawrence Livermore National Laboratory {schordan1,lin32,dquinlan}@llnl.gov
[2] University of California Los Angeles pouchet@cs.ucla.edu

Abstract. As processors gain in complexity and heterogeneity, compilers are asked to perform program transformations of ever-increasing complexity to effectively map an input program to the target hardware. It is critical to develop methods and tools to automatically assert the correctness of programs generated by such modern optimizing compilers.
We present a framework to verify if two programs (one possibly being a transformed variant of the other) are semantically equivalent. We focus on scientific kernels and a state-of-the-art polyhedral compiler implemented in ROSE. We check the correctness of a set of polyhedral transformations by combining the computation of a state transition graph with a rewrite system to transform floating point computations and array update operations of one program such that we can match them as terms with those of the other program. We demonstrate our approach on a collection of benchmarks from the PolyBench/C suite.

1 Introduction

The hardware trend for the foreseeable future is clear: symmetric parallelism such as in SIMD units is ubiquitous; heterogeneous hardware exemplified by System-on-Chips becomes the solution of choice for low-power computing; and processors' instruction sets keep growing with specialized instructions to leverage additional acceleration/DSP hardware introduced by manufacturers. This ever-increasing complexity of the computing devices is exacerbating the challenge of programming them: to properly harness the potential of a given processor, one has to significantly transform/rewrite an input program to match the features of the target hardware. Advanced program transformations such as coarse-grain parallelization, vector/SIMD parallelization, data locality optimizations, etc. are required to achieve good performance on a particular hardware. Aggressive optimizing compilers, as exemplified by *polyhedral compilation* [1], aim at automating these transformation stages to deliver a high-performance program that is transformed for a particular hardware. From a single input source, these compilers perform highly complex loop transformations to expose the proper grain of parallelism and data locality needed for a given processor. Such transformations include complex loop tiling and coarse-grain parallelization.

Polyhedral compilers have shown great promises in delivering high-performance for a variety of targets from a single input source (for example to map affine stencil computations for CPUs [2, 3], GPUs [4] and FPGAs [5]), where each target requires its dedicated set of program transformations. However, asserting the correctness of the generated code has become a daunting task. For example, on a 2D Finite-Difference

Time-Domain kernel, after transformation the loop bound expressions of the only parallel OpenMP `for` loop generated are about 15 lines long, making manual inspection out of reach. We also remark that verifying the polyhedral compiler engine itself, PoCC [6], which is the result of 8 years of multi-institution development is also out of reach: the compiler is actually around 0.5 million lines of codes, making the effort of producing a certification of these compiler optimizations in a manner similar to Leroy's Compcert work [7] extremely high.

In addition to high-level program transformations that are performed by the compiler, a series of low-level implementation choices can significantly challenge the design of a verification system. A compelling example relates to the floating point number implementation chosen by the back-end compiler. If one program is implemented using `double` precision (e.g., 64 bits) and the other program is implemented using for instance specialized 80 bits instructions, even if they are two totally equivalent programs in terms of semantics the output produced by these programs is likely to differ slightly: successive rounding and truncation effects will affect the output result, this even if the program is fully IEEE compliant.

We are in need for an automated system that asserts the correctness of the generated code by such optimizing compilers, in a manner that is robust to the back-end implementation decisions made by the compiler. Previous work such as Verdoolaege's [8] has focused on determining if two programs, in particular two *affine* programs [9], are semantically equivalent. Such tools require the input program (control flow and data flow) to be precisely modeled using combinations of affine forms of the surrounding loop iterators and program parameters. In contrast, our work uses a more practical approach with strong potential to be generalized to larger classes of programs. In the present work we focus on equivalence of affine programs where the value of all program parameters (e.g., problem size) is known at compile-time, with only simple data-dependent conditionals.

In this work, we propose an automated system to assert the equivalence of two affine programs, one of them being generated by the PolyOpt/C[1] compiler. At a high level, we combine the computation of a state transition graph with a rewrite system to transform the floating point operations and array update operations of one program such that we can match them (as terms) with those of the other program. We make the following contributions.

– We develop a new approach for determining the equivalence of two affine programs with simple data-dependent control-flow, leveraging properties of polyhedral optimizations to design a simple but effective rewriting system and equivalence checker.
– We provide extensive analysis of our method in terms of problem sizes and equivalence checking time.
– We evaluate our method on a relevant subset of polyhedral optimizations, asserting the correctness of PolyOpt/C on a collection of numerical kernels. Our work led to finding two bugs in PolyOpt/C, which were not caught with its current correctness checking test suite.

The rest of the paper is organized as follows. Sec. 2 describes polyhedral transformations and the class of programs analyzed in this paper. Sec. 3 describes our equivalence checking method. Sec. 4 provides extensive evaluation of our approach, asserting the correctness of PolyOpt/C for the tested benchmarks. Sec. 5 discusses related work, before concluding.

2 Polyhedral Program Transformations

Unlike the internal representation that uses abstract syntax trees (AST) found in conventional compilers, polyhedral compiler frameworks use an internal representation of imperfectly nested affine loop computations and their data dependence information as a collection of parametric polyhedra, this enables a powerful and expressive mathematical framework to be applied in performing various data flow analysis and code transformations. Significant benefits over conventional AST representations of computations include the effective handling of symbolic loop bounds and array index function, the uniform treatment of perfectly nested versus imperfectly nested loops, the ability to view the selection of an arbitrarily complex sequence of loop transformations as a single optimization problem, the automatic generation of tiled code for non-rectangular imperfectly nested loops [10, 2, 11], the ability to perform instancewise dataflow analysis and determine the legality of a program transformation using exclusively algebraic operations (e.g., polyhedron emptiness test) [9, 12], and more [13]. The *polyhedral model* is a flexible and expressive representation for loop nests with statically predictable control flow. Loop nests amenable to this algebraic representation are called *static control parts* (SCoP) [9, 13], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the enclosing loop iterators and variables that are constant during the SCoP execution (whose values are unknown at compile-time, a.k.a. program parameters). Numerous scientific kernels exhibit those properties; they can be found in image processing filters, linear algebra computations, etc.as exemplified by the PolyBench/C test suite [14].

In a polyhedral compiler, program transformations are expressed as a reordering of each *dynamic* instance of each syntactic statement in the program. The validity of this reordering is determined in PolyOpt/C by ensuring that the order in which each operations accessing the same array cell is preserved in the transformed code, this follows the usual definition of data dependence preserving transformations [15]. No transformation on the actual mathematical operations used during the computation is ever performed: each statement body has its structure and arithmetic operations fully preserved after loop transformations. Strength reduction, partial redundancy elimination, and other optimizations that can alter the statement body are not considered in the traditional parallelization/tiling polyhedral transformations [2] that we evaluate in PolyOpt/C. These properties allow the design of an analysis and a simple but effective rewriting rule system to proof equivalence, as shown in later Sec. 3.

In this work, we focus exclusively on polyhedral program variants that are generated by a polyhedral compiler, PolyOpt/C. Details on the variants considered are found in later Sec. 4. That is, the codes we consider for equivalence checking are *affine programs* that can be handled by PolyOpt/C. Technically, we consider a useful subset of affine programs where the loop bounds are fully computable at compile-time. That is, once the program has been transformed by PolyOpt/C (possibly containing program parameters, such as the problem/array sizes), the resulting program must have all parameters replaced by a numerical value, to ensure that loop bound expressions can be properly computed and analyzed by our framework. Looking at PolyBench/C benchmarks, a sample dataset is always provided, which implies that we know, at compile time, the value of all program parameters.

3 Approach

In our approach we verify that the sequence of update operations involving floating point operations on each array element in the original program is exactly the same as in an optimized version of the program. Our verification is a combination of static analysis, program rewrite operations, and a comparison based on an SSA form [16] where each array element is represented as a different variable (with its own SSA number).

Algorithm 1: DetermineFloatingPointAssignmentSequenceInSSA

```
Data: $P$ : Program
Result: $S_{ssa}$: sequence of floating point operations in SSA Form
$STG$=compute-STG($P$);
$A$=extract-floating-point-assignment-sequence($STG$);
foreach $a \in A$ do
    rewrite(a) [apply rewrite rules 1-11]
$S_{ssa}$=determineSSA(A);
```

Algorithm 2: Verify

```
Data: $P_1, P_2$ : Programs to verify
Result: $result$: true when Programs can be determined to be equivalent,
        otherwise false
$S_1$=DetermineFloatingPointAssignmentSequenceInSSA($P_1$);
$S_2$=DetermineFloatingPointAssignmentSequenceInSSA($P_2$);
$S'_1$=sort($S_1$);      – sort by unique lhs SSA variable of assignment
$S'_2$=sort($S_2$);      – sort by unique lhs SSA variable of assignment
if $match(S'_1,S'_2)$ then
    return true;
else
    return false;
```

In Algorithm 2 we use Algorithm 1 to determine the sequence of update operations for each program. Algorithm 1 first computes (statically) a state transition graph (STG). In the STG each node represents the state before an assignment or a condition. Edges represent state transitions. In our benchmark programs the loops have constant numeric bounds. We can therefore compute in each state a concrete value of each iteration variable. Floating point operations and updates on arrays are not evaluated. Next the (non-evaluated) operations on floating point variables are collected as a sequence of terms (function extract-array-element-assignment-sequence). We then apply 11 rewrite rules to normalize all extracted array updates. We remark that the `foreach` loop in Algorithm 1 is actually a parallel loop: each term rewriting can be computed independently of the others. On the normalized sequence of assignments we then determine an SSA Form.

Algorithm 2 matches the determined (normalized) floating point update sequences. Because these sequences are in SSA form, we can reorder them for the purpose of comparison. We sort each sequence and match the two sorted sequences of terms representing the assignments. If both sequences are equal, then the programs have been verified to perform an identical sequence of updates on floating point values. In the following sections we discuss each operation in detail.

3.1 Example

```
#pragma scop
for (t = 0; t < 2; t++) {
  for (i = 1; i < 16 - 1; i++)
    B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
  for (j = 1; j < 16 - 1; j++)
    A[j] = B[j];
}
#pragma endscop
```

Fig. 1. Original Jacobi-1d-Imper benchmark (only the loop is shown). The variable t is used for computing the number of steps and is set to 2 in the experiments. Array size is 16.

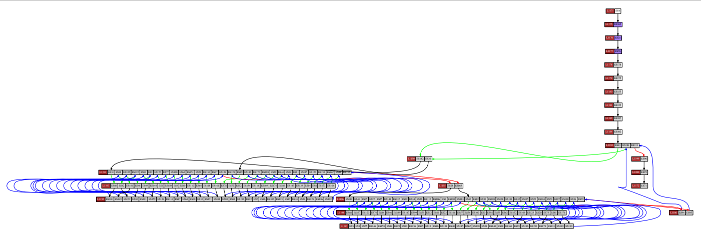
```
#pragma scop
{
  int c0;
  int c2;
  for (c0 = 1; c0 <= 17; c0++) {
    if (c0 >= 15) {
      if ((c0 + 1) % 2 == 0) {
        A[14] = B[14];
      }
    }
    for (c2 = (0 > (((c0 + -14) * 2 < 0?-(-(c0 + -14) / 2) : ((2 < 0?
          (-(c0 + -14) + - 2 - 1) / - 2 : (c0 + -14 + 2 - 1) / 2))))?
            0 : (((c0 + -14) * 2 < 0?-(-(c0 + -14) / 2) :
                ((2 < 0?(-(c0 + -14) + - 2 - 1) / - 2 : (c0 + -14 + 2 - 1) / 2)))));
         c2 <= ((1 < (((c0 + -2) * 2 < 0?
            ((2 < 0?-((-(c0 + -2) + 2 + 1) / 2) : -((-(c0 + -2) + 2 - 1) / 2))) :
            (c0 + -2) / 2))?1 : (((c0 + -2) * 2 < 0?
            ((2 < 0?-((-(c0 + -2) + 2 + 1) / 2) : -((-(c0 + -2) + 2 - 1) / 2))) : (c0 + -2) / 2))));
         c2++) {
        B[c0 + -2 * c2] = 0.33333 * (A[c0 + -2 * c2 - 1] + A[c0 + -2 * c2] + A[c0 + -2 * c2 + 1]);
        A[c0 + -2 * c2 + -1] = B[c0 + -2 * c2 + -1];
    }
    if (c0 <= 3) {
      if ((c0 + 1) % 2 == 0) {
        B[1] = 0.33333 * (A[1 - 1] + A[1] + A[1 + 1]);
      }
    }
  }
}
#pragma endscop
```

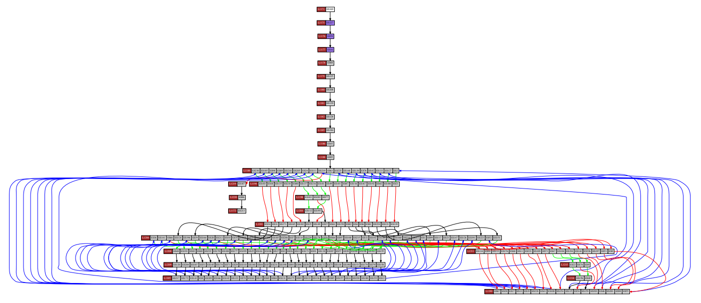
Fig. 2. Optimized Jacobi-1d-Imper benchmark ("tile_8_1_1" variant)

As running example we use the (smallest) benchmark Jacobi-1d-Imper. The original loop body is shown in Fig. 1 and an optimized variant ("tile_8_1_1" variant) is shown in Fig. 2. In the following sections we describe how to analyze and transform both programs to automatically verify that both programs are equivalent and the optimization performed by polyOpt/C is indeed semantics preserving and correct.

3.2 State Transition Graph Analysis

The state transition graph represents all possible states of a program. We use a symbolic representation of states where values and relations between variables are represented as predicates. If the concrete value of a variable is known, then all arithmetic operations are performed on this variable (without approximation). If no value is known (for example an input variable) then constraints are extracted from conditions and assignments, and a path-sensitive analysis if performed. The computation of the state transition graph (STG) has also been used in the RERS Challenge [17] 2012 and 2013 where the STG was then used for the verification of linear temporal logic formulas. For the verification of the polyhedra optimizations we use the STG to reason on the states of the program and extract the sequence of all array update operations that can be performed by the program. The implementation is integrated in our ROSE tool CodeThorn. Visualizations of the STG for our running example Jacobi-1d and one optimization variant are shown in Fig 3.2. The nodes in the graph represent program states, the edges represent state transitions. Currently we support the input language as represented by the C subset of the PolyBench/C programs for program equivalence checking.


Jacobi-1d-Imper (Original Program) - State Transition Graph


Jacobi-1d-Imper (Variant tile_8_1_1) - State Transition Graph

3.3 Floating Point Operation and Array Access Extraction

The floating point operation and array access extraction follows the reachability in the state transition graph (STG) from a selected node. In our case the selected node is the entry node of the function that contains the PolyOpt/C generated loop nest. These are marked in the PolyBench/C programs with pragmas (see Fig. 1 and 2).

Since we consider only the limited form of loops with constant numeric bounds, the entire state space of the loop can be computed in a form that is equivalent to loop unrolling. Our analyzer can also extract predicates from conditions for variables with unknown values. For the benchmarks this is not relevant though, because variables in those conditions have no data dependence on input values. The benchmarks contain conditions inside the loop body, guarding array updates, but those conditions only contain loop iteration variables. Since the values for the loop iteration variables are determined by the analysis (when computing the state transition graph), those conditions can be evaluated as well. Therefore, for the polyhedral benchmarks, our path sensitive analyzer can establish exactly one execution path for each given benchmark. From this determined state transition sequence we extract the terms of the floating point variable updates (including arrays). Only those terms representing variable updates and its corresponding state (containing a property state with a mapping of each iteration variable to a value) are relevant for the remaining phases.

3.4 Rewrite Rules

We establish a small set of rewrite rules which are sufficient to verify the given benchmarks. The rewrite rules operate on the terms of the program representing floating point operations and array updates (as described in Section 3.3).

1. $Minus(IntVal.val) \Rightarrow IntVal.val' = -IntVal.val$
2. $AddAssign(\$L, \$R) \Rightarrow Assign(\$L, Add(\$L, \$R))$
3. $SubAssign(\$L, \$R) \Rightarrow Assign(\$L, Sub(\$L, \$R))$
4. $MulAssign(\$L, \$R) \Rightarrow Assign(\$L, Mul(\$L, \$R))$
5. $DivAssign(\$L, \$R) \Rightarrow Assign(\$L, Div(\$L, \$R))$
6. $Add(Add(\$Remains, \$Other), IntVal) \Rightarrow Add(Add(\$Remains, IntVal), \$Other)$ where $\$Other \neq IntVal \land \$Other \neq FloatVal \land \$Other \neq DoubleVal$
7. $Add(IntVal_1, IntVal_2) \Rightarrow IntVal.val = IntVal_1.val\ +\ IntVal_2.val$
8. $Sub(IntVal_1, IntVal_2) \Rightarrow IntVal.val = IntVal_1.val\ -\ IntVal_2.val$
9. $Mul(IntVal_1, IntVal_2) \Rightarrow IntVal.val = IntVal_1.val\ *\ IntVal_2.val$
10. $Div(IntVal_1, IntVal_2) \Rightarrow IntVal.val = IntVal_1.val\ /\ IntVal_2.val$
11. If a variable v in term t has a constant value c in the associated state S in the STG, then replace the variable v with the constant c in the term t.

The rewrite rules are applied to each extracted assignment separately (i.e. each array assignment and floating point variable assignment). Rule 1 eliminates the unary minus operator. Rules 2-5 eliminate compound assignment operators that modify the same variable and replace it with the assignment operator. This step is a normalization step for the next phase were an SSA Form is established. Rule 6 requires some careful considerations. It performs a reordering of integer constants and is the only rule were we reorder expressions. Keeping in mind that we want to verify the property of the polyhedra generated code that ensures the lexicographic order is preserved, we do not want to reorder array updates. More specifically, we do not want to reorder any floating point operations. The above rule only reorders constant integer values such that we can establish new terms with one operator and two constants, suitable for constant folding by the rules 7-10. In Rule 11 the variables are replaced with the constant value that has been established in the state transition graph for that variable.

The rewrite rules are applied on each assignment representing an array update until no rule can be applied anymore. The rules guarantee termination (by design).

3.5 Verification

The verification steps consist of representing all indexed array elements as unique variables (each array element is considered as one unique variable), generating Static Single Assignment Form [16] for the sequence of array updates and floating point operations, and the final equality test (matching) of the SSA forms.

Represent each variable and array element as one variable and generate SSA Form. In Fig. 3 the final result for our running example Jacobi-1d is shown. The expressions of all assignments to arrays have been rewritten applying rules 1-11 (see Section 3.4). Each array element is treated as one separate variable. For example, `a[0]` is treated as a different variable to `a[1]`. The sequence of array operations (of the entire program extracted from the STG) is shown. For this sequence we establish SSA Form. The SSA numbers are post-fixed to each element. For example `B[1]=...;` `A[1]=B[1]` becomes `B[1]_1=...;` `A[1]_1=B[1]_1`. Note that the array notation is only for readability. At this stage it is only relevant to have a unique name for each memory cell (i.e. we could also rename the array element to `B_1_1`)

This step is similar with the compiler optimization of scalar replacement of aggregates (SRoA) and variables as also performed by LLVM after loop unrolling. In particular, LLVM also generates an SSA Form after this replacement. Thus, our approach is in this respect similar to existing analyses and transformations in existing compilers and may be suitable for a verifying compiler that also checks whether the transformed program preserves the program semantics w.r.t. the sequence of floating point operations. Note that SRoA is usually only applied up to a certain size of an aggregate as well.

The SSA numbering allows to define a set of assignments while preserving all data dependencies. If the sets are equal for two given programs, they are guaranteed to have the same sequence of updates and operations on all floating point variables independent from their values. Note that the benchmarks do contain conditionals inside the loop. But those can be completely resolved in the computation of the state transition graph when applied to integers for which constant values can be inferred. In cases where the value is unknown (e.g. a test on floating point values) the term remains in the expression; i.e. the analyzer performs a partial evaluation of the program and the non-evaluated part is represented as term. For the given benchmarks the SSA Form for the *extracted* sequence of updates does not require phi-assignments. The reason is that the benchmarks only contain conditionals on index variables which become constant after unrolling. Also see Fig. 2 for an example of a benchmark code with such properties. The ternary operator inside expressions is used inside floating point computations though. But since we represent in our analysis all floating point values to be an unknown value, the different states that are established by the analyzer during the expression analysis are determined to be equal, and thus, only a single state transition is established in the STG for such an assignment (see Fig. 4) and the extracted computation remains a sequence of assignments for the given programs. For each PolyBench/C 3.2 benchmark exactly one sequence of computations can be extracted.

Equality Test of SSA Forms. The final step in the verification is to determine the equivalence of the SSA Forms of two program variants. Our approach considers a verification to be successful if the term representations of the set of assignments in SSA Form is equal.

The interleaving of the assignments may differ as is demonstrated also in the example in Fig. 3, but the sequence of updates for each array element must be exactly the same. In particular, also the operations on the rhs of each assignment must match exactly (term equivalence). For the example in Fig. 3 this is indeed the case. For example the sequence of updates on the elements of `B[1]_1`, `B[2]_1` `B[3]_1` etc. is exactly the same in both columns. This holds for all SSA enumerated variables.

Jacobi-1D-Imper (original)	Jacobi-1D-Imper (Variant tile_8_1_1)
B[1]_1 = 0.33333 *(A[0]_0 + A[1]_0 + A[2]_0)	B[1]_1 = 0.33333 *(A[0]_0 + A[1]_0 + A[2]_0)
B[2]_1 = 0.33333 *(A[1]_0 + A[2]_0 + A[3]_0)	B[2]_1 = 0.33333 *(A[1]_0 + A[2]_0 + A[3]_0)
B[3]_1 = 0.33333 *(A[2]_0 + A[3]_0 + A[4]_0)	A[1]_1 = B[1]_1
B[4]_1 = 0.33333 *(A[3]_0 + A[4]_0 + A[5]_0)	B[3]_1 = 0.33333 *(A[2]_0 + A[3]_0 + A[4]_0)
B[5]_1 = 0.33333 *(A[4]_0 + A[5]_0 + A[6]_0)	A[2]_1 = B[2]_1
B[6]_1 = 0.33333 *(A[5]_0 + A[6]_0 + A[7]_0)	B[1]_2 = 0.33333 *(A[0]_0 + A[1]_1 + A[2]_1)
B[7]_1 = 0.33333 *(A[6]_0 + A[7]_0 + A[8]_0)	B[4]_1 = 0.33333 *(A[3]_0 + A[4]_0 + A[5]_0)
B[8]_1 = 0.33333 *(A[7]_0 + A[8]_0 + A[9]_0)	A[3]_1 = B[3]_1
B[9]_1 = 0.33333 *(A[8]_0 + A[9]_0 + A[10]_0)	B[2]_2 = 0.33333 *(A[1]_1 + A[2]_1 + A[3]_1)
B[10]_1 = 0.33333 *(A[9]_0 + A[10]_0 + A[11]_0)	A[1]_2 = B[1]_2
B[11]_1 = 0.33333 *(A[10]_0 + A[11]_0 + A[12]_0)	B[5]_1 = 0.33333 *(A[4]_0 + A[5]_0 + A[6]_0)
B[12]_1 = 0.33333 *(A[11]_0 + A[12]_0 + A[13]_0)	A[4]_1 = B[4]_1
B[13]_1 = 0.33333 *(A[12]_0 + A[13]_0 + A[14]_0)	B[3]_2 = 0.33333 *(A[2]_1 + A[3]_1 + A[4]_1)
B[14]_1 = 0.33333 *(A[13]_0 + A[14]_0 + A[15]_0)	A[2]_2 = B[2]_2
A[1]_1 = B[1]_1	B[6]_1 = 0.33333 *(A[5]_0 + A[6]_0 + A[7]_0)
A[2]_1 = B[2]_1	A[5]_1 = B[5]_1
A[3]_1 = B[3]_1	B[4]_2 = 0.33333 *(A[3]_1 + A[4]_1 + A[5]_1)
A[4]_1 = B[4]_1	A[3]_2 = B[3]_2
A[5]_1 = B[5]_1	B[7]_1 = 0.33333 *(A[6]_0 + A[7]_0 + A[8]_0)
A[6]_1 = B[6]_1	A[6]_1 = B[6]_1
A[7]_1 = B[7]_1	B[5]_2 = 0.33333 *(A[4]_1 + A[5]_1 + A[6]_1)
A[8]_1 = B[8]_1	A[4]_2 = B[4]_2
A[9]_1 = B[9]_1	B[8]_1 = 0.33333 *(A[7]_0 + A[8]_0 + A[9]_0)
A[10]_1 = B[10]_1	A[7]_1 = B[7]_1
A[11]_1 = B[11]_1	B[6]_2 = 0.33333 *(A[5]_1 + A[6]_1 + A[7]_1)
A[12]_1 = B[12]_1	A[5]_2 = B[5]_2
A[13]_1 = B[13]_1	B[9]_1 = 0.33333 *(A[8]_0 + A[9]_0 + A[10]_0)
A[14]_1 = B[14]_1	A[8]_1 = B[8]_1
B[1]_2 = 0.33333 *(A[0]_0 + A[1]_1 + A[2]_1)	B[7]_2 = 0.33333 *(A[6]_1 + A[7]_1 + A[8]_1)
B[2]_2 = 0.33333 *(A[1]_1 + A[2]_1 + A[3]_1)	A[6]_2 = B[6]_2
B[3]_2 = 0.33333 *(A[2]_1 + A[3]_1 + A[4]_1)	B[10]_1 = 0.33333 *(A[9]_0 + A[10]_0 + A[11]_0)
B[4]_2 = 0.33333 *(A[3]_1 + A[4]_1 + A[5]_1)	A[9]_1 = B[9]_1
B[5]_2 = 0.33333 *(A[4]_1 + A[5]_1 + A[6]_1)	B[8]_2 = 0.33333 *(A[7]_1 + A[8]_1 + A[9]_1)
B[6]_2 = 0.33333 *(A[5]_1 + A[6]_1 + A[7]_1)	A[7]_2 = B[7]_2
B[7]_2 = 0.33333 *(A[6]_1 + A[7]_1 + A[8]_1)	B[11]_1 = 0.33333 *(A[10]_0 + A[11]_0 + A[12]_0)
B[8]_2 = 0.33333 *(A[7]_1 + A[8]_1 + A[9]_1)	A[10]_1 = B[10]_1
B[9]_2 = 0.33333 *(A[8]_1 + A[9]_1 + A[10]_1)	B[9]_2 = 0.33333 *(A[8]_1 + A[9]_1 + A[10]_1)
B[10]_2 = 0.33333 *(A[9]_1 + A[10]_1 + A[11]_1)	A[8]_2 = B[8]_2
B[11]_2 = 0.33333 *(A[10]_1 + A[11]_1 + A[12]_1)	B[12]_1 = 0.33333 *(A[11]_0 + A[12]_0 + A[13]_0)
B[12]_2 = 0.33333 *(A[11]_1 + A[12]_1 + A[13]_1)	A[11]_1 = B[11]_1
B[13]_2 = 0.33333 *(A[12]_1 + A[13]_1 + A[14]_1)	B[10]_2 = 0.33333 *(A[9]_1 + A[10]_1 + A[11]_1)
B[14]_2 = 0.33333 *(A[13]_1 + A[14]_1 + A[15]_0)	A[9]_2 = B[9]_2
A[1]_2 = B[1]_2	B[13]_1 = 0.33333 *(A[12]_0 + A[13]_0 + A[14]_0)
A[2]_2 = B[2]_2	A[12]_1 = B[12]_1
A[3]_2 = B[3]_2	B[11]_2 = 0.33333 *(A[10]_1 + A[11]_1 + A[12]_1)
A[4]_2 = B[4]_2	A[10]_2 = B[10]_2
A[5]_2 = B[5]_2	B[14]_1 = 0.33333 *(A[13]_0 + A[14]_0 + A[15]_0)
A[6]_2 = B[6]_2	A[13]_1 = B[13]_1
A[7]_2 = B[7]_2	B[12]_2 = 0.33333 *(A[11]_1 + A[12]_1 + A[13]_1)
A[8]_2 = B[8]_2	A[11]_2 = B[11]_2
A[9]_2 = B[9]_2	A[14]_1 = B[14]_1
A[10]_2 = B[10]_2	B[13]_2 = 0.33333 *(A[12]_1 + A[13]_1 + A[14]_1)
A[11]_2 = B[11]_2	A[12]_2 = B[12]_2
A[12]_2 = B[12]_2	B[14]_2 = 0.33333 *(A[13]_1 + A[14]_1 + A[15]_0)
A[13]_2 = B[13]_2	A[13]_2 = B[13]_2
A[14]_2 = B[14]_2	A[14]_2 = B[14]_2

Fig. 3. Example: Extracted assignments (updates) from the programs in Fig. 1 (left column) and Fig. 2 (right column) after rewrite and renaming in SSA Form. The rewrite rules that are applied are those listed in Section 3.4. Rewrite statistics for this example are shown in Table 2 (see rows for jacobi-1d-imper and jacobi-1d-imper-tile-8-1-1). The number of extracted assignments is 56 and the number of applied rewrite operations differs significantly, but the final results are two sequences of assignments that are equivalent - they do differ in the interleaving of the assignments, but the order of updates on all SSA variables is identical. This is checked by Algorithm 2 after sorting both sequences by the updated SSA variable.

For all the evaluated benchmarks the extracted SSA Forms match exactly as unordered sets. The small set of rewrite rules presented in Section 3.4 is sufficient to proof equivalence for the Polybech/C 3.2 benchmarks.

Supported Language Subset. In Fig. 4 a fragment of the update sequence for the correlation benchmark is shown. It includes the use of a floating point variable which is assigned a value outside the polyhedral optimized program section and therefore has SSA number 0, the ternary operator, an external function call (sqrt), and a computation involving different arrays and their elements. *This code does not have a static control-flow*, because of the ternary operator leading to a data-dependent assignement of the value. Previous work on affine program equivalence [8] cannot handle such case, in contrast our approach supports such construct.

```
stddev[0]_18 = stddev[0]_17 / float_n_0
stddev[0]_19 = sqrt(stddev[0]_18)
stddev[0]_20 =(stddev[0]_19 <= eps_0?1.0 : stddev[0]_19)
stddev[1]_1 = 0.0
stddev[1]_2 = stddev[1]_1 +(data[0][1]_0 - mean[1]_18) *(data[0][1]_0 - mean[1]_18)
```

Fig. 4. Fragment of verified update sequence for datamining/correlation benchmark.

For a defined set of external function calls we assume that the functions are side effect free (e.g. sqrt). For each PolyBench/C benchmark and PolyOpt/C generated variant with constant array bounds, our analysis can determine an STG with exactly one floating point computation sequence. We consider cases where more than one execution path is represented in the STG in our future work.

Error Detection. When the equality test fails, then the semantic equivalence of two programs cannot be established. This can have two reasons i) the programs are different, ii) our rewrite system is not powerful enough to establish a normalized representation such that the two programs' sequence of floating-point operations can be matched. For two benchmarks we determined differences in the update sequence (cholesky and reg_detect), as shown in Sec. 4.3. The difference is reported as the set of non-matching assignments.

4 Results

Our implementation is based on ROSE [18]. The computation of the state transition graph (STG) has also been used in the RERS Challenge [17] 2012 and 2013.

The rewrite system is based on the AstMatching mechanism in ROSE and implements the small number of rules that turned out to be sufficient to verify benchmarks of the PolyBench/C suite.

Benchmarks in PolyBench/C 3.2 contain SCoPs and represent computation in linear algebra, datamining, and stencil computing. PolyOpt/C performs data dependence analysis, loop transformation and code generation based on the polyhedral model. Because of the transformation capability and its integration in the ROSE compiler, PolyOpt/C is chosen as the optimization driver in the experiments. Optimization variants in this study are mainly generated from the following two transformations:

– Tiling-driven transformations: the ”–polyopt-fixed-tiling” option in PolyOpt/C implements arbitrarily complex sequences of loop transformations to maximize the tilability of the program, applies tiling, and expose coarse-grain or wavefront parallelism between tiles [2]. It allows to specify the tile size to be used in each tiled dimension.
– Data locality-driven transformations: we use the Pluto algorithm [2] for maximal data locality, and test the three different statement fusion schemes (minfuse, maxfuse and smartfuse) implemented in PolyOpt/C.

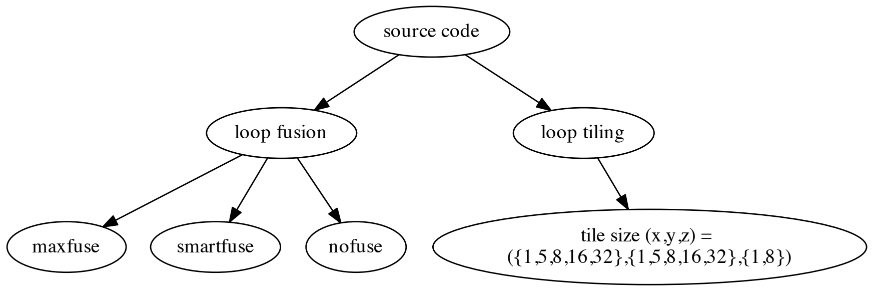


Fig. 5. Transformation variants: 53 variants are generated for each benchmark (3 for fusion and $5 \times 5 \times 2 = 50$ for tiling).

Figure 5 illustrates the transformation flow. Each benchmark code with a constant array size (size in each dimension) is given to PolyOpt/C. The following variants are generated for the study:

1. Loop fusion: Polyhedral transformation performs multiple transformations in a single phase. In this variant, we apply the three fusion schemes: maxfuse, smartfuse and nofuse to drive the loop transformation. A combination of loop permutation, skewing, shifting, fusion and distribution is implemented for each of the three statement fusion schemes.
2. Loop tiling: PolyOpt/C takes 3 parameters to form a tile size for single and multi-dimensional tiling. Tile sizes with number of power of two are commonly seen in real applications. Other special tile sizes, such as size in prime numbers, or non-tilable sizes (size larger than problem/array size), are also included to prove a broader span for verification: the polyhedral code generator CLooG [11] integrated in PolyOpt/C generates a possibly different code for each of these cases. Some tile sizes might not be applicable to all benchmarks (e.g., those which cannot be tiled along multiple dimensions), but PolyOpt/C still generates a valid output with 0D or 1D tiling for the verification. Note that we also want to verify certain corner cases in the code generation.

With our approach we can verify all PolyOpt/C generated optimization variants for PolyBench/C 3.2. As shown in Table 1) PolyOpt/C generated 53 variants for all but two benchmarks. Unfortunately, for 'doitgen' only the 3 fusion variants could be generated, and for dynprog the fusion and tiling variants could not be generated (due to an error in PolyOpt/C 0.2). For all other benchmarks all variants were generated (in total 53×28). We verified $53 \times 28 + 3 = 1487$ variants, and found errors in the fusion (2 of 3) and the tiling variants of the cholesky benchmark and errors in the tiling variants of 'reg_detect' (in total $2 + 25 + 25 = 104$). The array size is set to 16 for each dimension in the 1D ,2D, and 3D cases and stepsize is set to 2 (stepsize is the time dimension for stencil benchmarks). The total verification time for all 1487 variants is less than 2 hours, as shown in column 3 in Table 1.

We recall a critical aspect of affine programs is that they have a control flow that only depends on loop iterators and program parameters. That is, while only one C program is generated by PolyOpt/C, it is by construction expected to be valid for any value the program parameters can take (e.g., the array size `N`). Checking the correctness of this code for a small array size (e.g., `N` = 16) still stresses the entire code for the benchmarks we considered.

Benchmark	Verification (Size 16)	Total Run Time
2mm, 3mm, adi, atax, bicg, covariance, durbin, fdtd-2d, gemm, gemver, gesummv, jacobi-1d, jacobi-2d, lu, ludcmpm, mvt, seidel, symm, syr2k, syrk, trisolv, trmm, correlation, gramschmidt, fdtf-ampl, floyd-warshall	for 26 benchmarks all 53 variants verified	1h:48m:41s
dotitgen	all fusion variants verified	
cholesky	errors found in fusion and tiling opt	
reg_detect	errors found in tiling opt	
doitgen	tiling variants not available	-
dynprog	variants not available	-

Table 1. The benchmarks in the Polybench/C 3.2 suite with information whether we successfully verified the PolyOpt/C generated variants. For two benchmarks our verification procedure helped to find bugs in fusion and tiling optimizations in PolyOpt/C 0.2

4.1 State Space and Rewrite Operations Statistics

In Table 2 detailed statistics on some of the benchmarks are shown. The statistics include the number of computed states in the state transition graph and how often each of the rewrite rule has been applied for each benchmark variant. The original program is listed by the name of the benchmark itself. Optimization variants are denoted by the benchmark name prefixed with the variant name.

4.2 Run Times

The run times for each benchmark and each generated optimization variant is shown in the last column in Table 2. This includes all phases (parsing, STG analysis, update extraction, update normalization, and sorting of the update sequence). The total verification time of an original program and a variant is the sum of the total time (as shown in Table 2 of both entries in the table plus the time for comparison. The final comparison is linear in the number of assignments because Algorithm 1 also includes sorting by the unique SSA assignment variable on lhs of each update operation.

For example, to verify that the benchmark `jacobi-1d-imper` and the polyhedral optimization variant `tile_8_1_1` are equivalent, the total verification time is the sum of the run times for the original benchmark and the variant (each one shown in last column in Table 2) and the time for matching the assignments of the two sorted lists of assignments (not shown). Note that this is also valid for any pair of variants for the same benchmark, hence the original benchmark only needs to be analyzed once. The total run time for the verification of all 1487 generated benchmark variants, including all operations, is shown in Table 1.

Benchmark-Variant	States	Updates	R1	R2-5	R6	R7-10	R11	Run Time
3mm	40922	13056	0	12288	0	0	75264	6.13 secs
3mm-fuse-smartfuse	40925	13056	0	12288	0	0	75264	6.17 secs
3mm-tile-8-1-1	50916	13056	0	12288	0	0	75264	6.53 secs
covariance	9125	2992	0	2704	0	0	15440	1.56 secs
covariance-fuse-smartfuse	9128	2992	0	2704	0	0	15440	1.57 secs
covariance-tile-8-1-1	12652	2992	0	2704	0	0	15440	1.71 secs
fdtd-2d	4731	1442	0	0	0	1860	13144	1.58 secs
fdtd-2d-fuse-smartfuse	4734	1442	0	0	0	1860	13144	1.59 secs
fdtd-2d-tile-8-1-1	5436	1442	17700	0	6090	28260	21356	3.14 secs
fdtd-apml	52829	34816	0	0	0	12544	353792	25.16 secs
fdtd-apml-fuse-smartfuse	52832	34816	0	0	0	12544	353792	25.39 secs
fdtd-apml-tile-8-1-1	60171	34816	0	0	0	12544	353792	25.49 secs
floyd-warshall	13388	4096	0	0	0	0	57344	3.14 secs
floyd-warshall-fuse-smartfuse	13391	4096	0	0	0	0	57344	3.20 secs
floyd-warshall-tile-8-1-1	15776	4096	0	0	0	0	57344	3.30 secs
gramschmidt	14072	4504	0	2176	0	0	29712	2.26 secs
gramschmidt-fuse-smartfuse	14120	4504	0	2176	0	0	29712	2.29 secs
gramschmidt-tile-8-1-1	18924	4504	0	2176	0	0	29712	2.43 secs
jacobi-1d-imper	194	56	0	0	0	56	168	369.76 ms
jacobi-1d-imper-fuse-smartfuse	196	56	0	0	0	56	168	364.77 ms
jacobi-1d-imper-tile-8-1-1	232	56	208	0	78	420	312	393.33 ms
jacobi-2d-imper	2602	784	0	0	0	1568	6272	692.29 ms
jacobi-2d-imper-fuse-smartfuse	2605	784	0	0	0	1568	6272	696.55 ms
jacobi-2d-imper-tile-8-1-1	3923	784	7192	0	1560	15032	11680	1.50 secs
seidel-2d	1309	392	0	0	0	4704	7840	1.05 secs
seidel-2d-fuse-smartfuse	1312	392	0	0	0	4704	7840	1.08 secs
seidel-2d-tile-8-1-1	2144	392	11760	0	2352	28224	19600	2.51 secs
trmm	6794	1920	0	1920	0	0	11520	1.29 secs
trmm-fuse-smartfuse	6797	1920	0	1920	0	0	11520	1.31 secs
trmm-tile-8-1-1	9438	1920	3840	1920	0	7680	15360	2.52 secs

Table 2. Shows from left to right for some selected benchmark results: the number of computed states in the STG, number of extracted assignments (updates), how often the rewrite rules 1-11 were applied, and the run time.

4.3 Bugs Found Thanks to Verification

While our verification asserted the correctness of 1383 different programs variants generated by PolyOpt/C for the tested problem sizes, a significant outcome is the finding of two previously unknown bugs, for cholesky and reg_detect. For cholesky we determined that one assignment pair (for `A[1][0]_1`) does not match. All other 951 assignments do match (see Fig. 6).

Benchmark	Original Program	Variant tile_8_1_1 (Detected Errors)
cholesky	`A[1][0]_1 = x_2 * p[0]_1`	`A[1][0]_1 = x_1 * p[0]_1`
reg_detect	`mean[0][0]_1 = sum_diff[0][0][15]_1`	`mean[0][0]_1 = sum_diff[0][0][15]_0`
	`mean[0][0]_2 = sum_diff[0][0][15]_2`	`mean[0][0]_2 = sum_diff[0][0][15]_1`
	`mean[0][1]_1 = sum_diff[0][1][15]_1`	`mean[0][1]_1 = sum_diff[0][1][15]_0`
	`mean[0][1]_2 = sum_diff[0][1][15]_2`	`mean[0][1]_2 = sum_diff[0][1][15]_1`
	`mean[1][1]_1 = sum_diff[1][1][15]_1`	`mean[1][1]_1 = sum_diff[1][1][15]_0`
	`mean[1][1]_2 = sum_diff[1][1][15]_2`	`mean[1][1]_2 = sum_diff[1][1][15]_1`

Fig. 6. Errors found in generated optimized programs. The equality check in Algorithm 2 reported semantic inequality because the right-hand-sides of some corresponding assignments are different. Shown are those floating-point operations that do not match.

The current test suite of PolyOpt/C checks the correctness of the transformed code by checking if the output of the computation is strictly identical for the reference and transformed codes. Under this scheme, errors that amount to changing the order of two update operations (thereby violating the dependence between such operations) may not be caught: in practice, IEEE floating point operations are often commutative/associative and therefore changing the order of their computation may not always lead to a different final result. The clear merit of our approach is demonstrated by finding two bugs that could hardly be caught by classical testing means, but was immediately found by our verification process. In addition, the ability to point to the set of operations that do not match greatly helps the bug finding process.

5 Related Work

Existing research adopts various approaches to verify the transformation results. Focusing on affine programs, Verdoolaege et al. develop an automatic equivalence proofing [8]. The equivalence checking is heavily requiring the fact that input programs have an affine control-flow, as the method is based on mathematical reasoning about integer sets and maps built from affine expressions, and the development of widening/narrowing operators to properly handle non-uniform recurrences. In contrast, our work has a strong potential for generalization beyond affine programs. In fact, we already support some cases of data-dependent control-flow in the verification, something not supported by previous work [8].

Karfa et al. also designed a method exclusively for a subset of affine programs, using array data dependence graphs (ADDGs) to represent the input and transforming behaviors. An operator-level equivalence checking provides the capability to normalize the expression and establish matching relations under algebraic transformations [19]. Mansky and Gunter [20] use the TRANS language [21] to represent transformations. The correctness proof is verified by Isabelle [22], a generic proof assistant, implemented in the verification framework.

6 Conclusion

We have presented an approach for verifying that the implementation of PolyOpt/C for polyhedral optimizations is semantics preserving. Our approach first performs a static analysis and determines a list of terms representing the updates on floating point variables and array elements. This sequence is then rewritten by a rewrite system and eventually SSA Form is established were each array element is treated as a separate variable. When the sets of array updates are equal and all terms match exactly then we have determined that the programs are indeed semantically equivalent. Otherwise we do not know whether the programs are equivalent or not. With our approach we were able to verify all PolyOpt/C 0.2 generated variants for PolyBench/C 3.2, out of which 1383 variants were shown to be correct, and we found errors in 104 generated variants, corresponding to one bug occuring for two benchmarks. This bug was not previously known and was not caught by the existing test suite of PolyOpt/C, which is based only on checking that the output data produced by a transformation is identical to the output produced by the reference code. We limited our evaluation to a size of 16 (for each array dimension) because our approach requires to analyze the entire state space of the loop iterations and we wanted to keep the overall verification time for all benchmarks and variants within a few hours, such that the verification procedure can be used in the release process of PolyOpt/C in future.

References

1. Pouchet, L.N.: PolyOpt/C 0.2.0: a Polyhedral Compiler for ROSE. http://www.cs.ucla.edu/ pouchet/software/polyopt/ (2012)
2. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. (June 2008)
3. Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.N., Sadayappan, P.: When polyhedral transformations meet simd code generation. In: PLDI. (June 2013)
4. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on gpu architectures. In: ICS. (June 2012)
5. Pouchet, L.N., Zhang, P., Sadayappan, P., Cong, J.: Polyhedral-based data reuse optimization for configurable computing. In: FPGA. (February 2013)
6. Pouchet, L.N.: PoCC 1.2: the Polyhedral Compiler Collection. http://www.cs.ucla.edu/ pouchet/software/pocc/ (2012)
7. Leroy, X.: The Compcert C compiler. http://compcert.inria.fr/compcert-C.html (2014)
8. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. ACM Transactions on Programming Languages and Systems (TOPLAS) **34**(3) (2012) 11
9. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: multidimensional time. Intl. J. of Parallel Programming **21**(6) (December 1992) 389–420
10. Irigoin, F., Triolet, R.: Supernode partitioning. In: ACM SIGPLAN Principles of Programming Languages. (1988) 319–329
11. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04), Juan-les-Pins, France (September 2004) 7–16
12. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: Convexity, pruning and optimization. In: POPL. (January 2011) 549–562
13. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations. Intl. J. of Parallel Programming **34**(3) (June 2006) 261–317
14. Pouchet, L.N.: PolyBench/C 3.2. http://www.cs.ucla.edu/ pouchet/software/polybench/ (2012)
15. Allen, J., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers (2002)
16. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13**(4) (Oct 1991) 451–490
17. Steffen (Organizer), B.: RERS Challenge: Rigorous Examination of Reactive Systems. http://www.rers-challenge.org (2010, 2012, 2013, 2014)
18. Quinlan, D., Liao, C., Matzke, R., Schordan, M., Panas, T., Vuduc, R., Yi, Q.: ROSE Web Page. http://www.rosecompiler.org (2014)
19. Karfa, C., Banerjee, K., Sarkar, D., Mandal, C.: Verification of loop and arithmetic transformations of array-intensive behaviors. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **32**(11) (2013) 1787–1800
20. Mansky, W., Gunter, E.: A framework for formal verification of compiler optimizations. In: Interactive Theorem Proving. Springer (2010) 371–386
21. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(4) (2009) 14
22. Paulson, L.C.: Isabelle Page. https://www.cl.cam.ac.uk/research/hvg/Isabelle