ECE1754 Project Report

Behrooz Zarebavani

April 26, 2021

1 Abstract

Compiler transformations, especially loop transformations, play a significant role in improving the runtime of many time-intensive kernels. As a result, guaranteeing the correctness of these transformations play an important role. There are three major approaches to prove the correctness of the program. I) proof of equivalence of affine programs, II) Matching the execution trace, and III) checking the output of the program via testing. All of these approaches have their flaws, and Polycheck attempt to solve these problems by combining a runtime check with the power of trace checking. In what follows, I will implement a demo of their first algorithm (Algorithm A in the paper) and evaluate a transformation to see whether it can detect errors that the testbench that I am using cannot detect.

2 Introduction

When a code is transformed, typically, it is not like the original code at all. Experts use automatic testing via a choice of a dataset to prove that their transformation is correct. However, in many cases, this may backfire. Imagine a matrix-matrix multiplication with zero matrices as an input. Thus, any code that generates zero az output can be assumed to be correct.

For addressing this data-dependence problem, automatic equivalence checking techniques such as static verification are introduced. This technique proved to be hard to use and requires a solid mathematical foundation and programming techniques. Also, it is constrained by the type of program that it can verify. For example, if the program is non-affine, it will have a hard time proving the code's correctness.

To address this problem, that both codes should be affine, experts come up with trace-based matching. That is, the trace of the two programs should be matched entirely. An example of this technique is in [1]. The disadvantage of this method is the space of the trace. Usually, this space is enormous and requires a significant amount of time to be checked and generated (please consider a huge graph matching problem). Polycheck [2] tries to address these problems by combining the trace idea with the runtime check. It instruments the code with some assertions which identify any deviation in the data-dependency flow. After retrieving their code, I realized that they use the PoCC compiler, which is the result of many previous scientific works like [3] and an integer set programming library, namely ISL [4],

combined with ROSE [5] compiler. Since it was hard for me to learn all of them (and I was in the middle of my implementation), I decided to choose the ROSE entirely. Thus, since it was my first attempted to get familiar with a compiler, I believe that I missed multiple corner cases, so this work is limited. But it was a great experiment for me to learn and think of the code the way a compiler is seeing it. After some simplification, in this work, I am presenting a ROSE pass that reads the code. Using some predefined libraries, it will instrument the code and verify the correctness of a transformation. The milestones are as follow:

- Implementation of four general function (firstWriter, nextWriter, writeBeforeRead, and lastWriter) and infrastructure to work with these functions resulted in polycheck_demo_functions.h header file.
- Computation of the original schedule of the code [6]
- Computation of the required mapping between dataspace and iteration space using ROSE compiler
- Instrumentation of the code with right helper variables, initialization of these variables and their usage using ROSE compiler

3 Methods

A transformed program is declared to be equivalent to an input program if I) the statement instances in the transformed program can be mapped in a bijective, or one-to-one, fashion to statement instances in the input program, and II) each statement instance in such a bijection satisfies the same dependencies as in the input program.

Since I simplify the problem to use only one statement, we can assume that we only have to check the dependencies constraints. To do that, we assign a shadow variable to each data space of the program. These shadow variables will store the last statement instance that write to their corresponding data space. Then, we can use this information and the original schedule of the input program to check the dependency constraints.

To clarify, please consider the original code in listing 1. After transforming the code using a recursive function that imitates the tiling approach, we will have the transformed code in listing 2. Now, we first have to do a light analysis on the original code. This analysis involves computing the original schedule used as the statement instance form $S < t, i, j >$. Please note that the schedule computation describes in the implementation methods works for arbitrary code and an arbitrary number of statements. However, for the use in this demo, I will abstract that general view that is $S < 0, t, 0, i, 0, j, 0 >$ into the $S < t, i, j >$.

After original schedule computation, we will drive some mapping between iteration space and data space. For example a mapping like $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right) \times \left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) + \left(\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right)$ drives the data reference subscripts of A[i - 1][j].

```
void Original_Sidel()
   for(int t = 0; t < T; t++)
      for(int i = 1; i < N; i++)
         for(int j = 1; j < N; j++)
            A[i][j] = A[i - 1][j] + A[i][j - 1];
```

Listing 1: The original code

After driving these mapping and the original schedule, we will instrument the Transformed_Sidel function with specific assertion to check whether the code complies to data dependency or not.

```
void Transformed_Sidel(int t,int ilo,int ihi, int jlo, int jhi){
   if (ilo > ihi || jlo > jhi) return;
   if (ilo == ihi && jlo == jhi){
      A[ilo][jlo] = A[ilo-1][jlo] + A[ilo][jlo-1];
   }else{
      Transformed_Sidel(t, ilo,(ilo + ihi) / 2, jlo, (jlo + jhi)/2); //Top-
   Left
      Transformed_Sidel(t, ilo,(ilo + ihi) / 2, (jlo + jhi) / 2 + 1,jhi); //
   Top-Right
      Transformed_Sidel(t, (ilo + ihi) / 2 + 1, ihi, jlo, (jlo + jhi) / 2);
   //Bottom-Left
      Transformed_Sidel(t, (ilo + ihi) / 2 + 1, ihi, (jlo + jhi)/2 + 1, jhi)
   ; //Bottom-Right
   }
   if (ilo == 1 && ihi == N - 1 && jlo == 1 && jhi == N-1 && t < (T - 1)){
      Transformed_Sidel(t+1, ilo, ihi, jlo, jhi);// Next time step
   }
}
```

Listing 2: The transformed code

To describe this part, lets assume that an unknown statement instance in the transformed code will result in following statement A[a][b] = A[c][d] + A[f][h]. First, using the shadow(A[a][b]) and (a,b) we will map the current statement instance to a statement instance in the original code. For example assume that a = 5, b = 7 and the last statement instance that write to this address, namely A[5][7], stored in shadow(A[a][b]) is $S < 1, 5, 7 >$. Based on the original schedule and (a,b) we can determine that the current statement instance should be $S < 2, 5, 7 >$.

Now using the current statement instance, we will regenerate the whole statement instance that implies $S < 2, 5, 7 >$. That is using the mapping that we derived in compile time we will generate the following statement instance A[5][7] = A[4][7] + A[5][6]. At this point we should check whether the current statement A[a][b] = A[c][d] + A[f][h] is valid or not. That is assert(c == 4), assert=(d == 7), assert(f == 5) and assert(h == 6).

After verifying that the statement is reading from the right places and write into the correct space, we need to check the values of the read data spaces. We should check whether the last statement instance that writes into these places is valid. To do that, we again get the current statement instance computed from the previous step, which is $S < 2, 5, 7 >$. Then passing the read data space and this statement instance to a function f($S < 2, 5, 7 >, (c, d)$), we can drive the last statement instance that was supposed to write in A[c][d], which is before $S < 2, 5, 7 >$. In here, it would be $S < 2, 4, 7 >$. Then we will check this with the shadow(A[c=4][d=7]). If they were the same, the data is valid. We also do the same thing for A[f][h].

That is the basic idea. The rest of the implementation details will be describe in the Implementation section.

4 Implementation

4.1 Main Function

I will start to explain the whole implementation, using the demo code, and function by function. So, for start, please see the main function in listing 4.

4.1.1 Compiler code structure

```
void PolyCheckInstrumentation::startInstrumenting(){
//Add Header
addInstrumentationHeader();
//Add Variables definitions
definePolyCheckGlobalVariables();
//initialize the Variables (extracting the statement information and
initialize the shadow variable)
initPolyCheckVariables();
//Also add required assertions
instrumentCode();
}
```

Listing 3: The core function that does the instrumentation in ROSE code

Since the report has limited space, I will briefly explain the core compiler implementations. Please see listing 3. The instrumentation class in my code, has 4 main functions. the "addInstrumentationHeader" will add the polycheck_demo_functions.h which contain the 4 functions that I am using for data dependency checks. For that, I am using "SageInterface::insertHeader" and I add this header into the global scope.

Then I need to define some global variables using "definePolyCheckGlobalVariables". To do that I am using "SageInterface::addTextForUnparser" interface and add the variables into the global scope.

Now the most important function in this implementation is "initPolyCheckVariables" which initialize the require variables by analyzing the input code. The implementation details is a bit repetitive. So to simplify, I basically generate a simple AST graph with a toy example. Then I just see the required nodes characteristics like their parents, the operations that I allowed them to take, etc. To do that, I have used "NodeQuery::querySubTree" interface extensively and iterate over the nodes inside the tree. I also implemented some general codes to attach original schedule and print statements. However, later, I find out that functions "unparseToString" exist to just implement these for me. Also, at some point, I noticed that generalization is just getting worse when I proceed, so I end-up leaving some of my general implementations and stick to the simplified one. Because there where no use of them.

```
int main() {
   //======================== Compiler stuff ========================
   std::vector<std::vector<bool>> readWriteSet(N, std::vector<bool>(N, true)
   );//True for read set and False for the write set

   //Initialize Shadow variables
   for(int i = 1; i < N; i++){
      for(int j = 1; j < N; j++){
         //WriteSet
         if(readWriteSet[i][j]){//Write ref
            shadow[i][j].invalidate();
            shadow[i][j].makeNoInit();
            readWriteSet[i][j] = false;
         }

         if(readWriteSet[i - 1][j]){//Read ref
            shadow[i - 1][j].makeValid();
            shadow[i - 1][j].makeInit();
            readWriteSet[i - 1][j] = false;
         }

         if(readWriteSet[i][j - 1]){//Read ref
            shadow[i][j - 1].makeValid();
            shadow[i][j - 1].makeInit();
            readWriteSet[i][j - 1] = false;
         }
      }
   }

   //Initialize Mapping
   mapping.LHS_MAP.push_back(std::vector<int> {1, 0}); // Mapping for first
   subscript
   mapping.LHS_MAP.push_back(std::vector<int> {0, 1}); // Mapping for second
```

```
    subscript

    mapping.RHS_MAP.push_back(std::vector<int>({0, 1, 0}));
    mapping.RHS_MAP.push_back(std::vector<int>({0, 0, 1}));
    mapping.RHS_MAP.push_back(std::vector<int>({0, 1, 0}));
    mapping.RHS_MAP.push_back(std::vector<int>({0, 0, 1}));

    //Compute Bounds
    min_bound.instance.resize(3);
    min_bound.instance[0] = 0;
    min_bound.instance[1] = 1;
    min_bound.instance[2] = 1;
    min_bound.makeValid();

    max_bound.instance.resize(3);
    max_bound.instance[0] = T;
    max_bound.instance[1] = N;
    max_bound.instance[2] = N;
    max_bound.makeValid();

    //Compute wref related invariants
    wref_fix_flag.push_back(false);
    wref_fix_flag.push_back(true);
    wref_fix_flag.push_back(true);

    //Extract constants from the read reference
    read_const.push_back(-1);
    read_const.push_back(0);
    read_const.push_back(0);
    read_const.push_back(-1);
    //============================ Test Section ======================
    //Generate some random data for Seidel
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(1.0, 10.0);
    for(int i = 0; i < N; i++){
       for(int j = 0; j < N; j++){
          Original_A[i][j] = dist(mt);
          Transformed_A[i][j] = Original_A[i][j];
          Instrument_A[i][j] = Original_A[i][j];
       }
    }
    Original_Seidel();
    Transformed_Seidel(0, 1, N - 1, 1, N - 1);
    Instrumented_Seidel(0, 1, N - 1, 1, N - 1);
```

```
76    for(int i = 0; i < N; i++){
77       for(int j = 0; j < N; j++){
78          if(Original_A[i][j] != Transformed_A[i][j])
79             std::cout << "There␣is␣a␣problem␣in␣testing" << std::endl;
80       }
81    }
82
83    //======================== Compiler stuff ========================
84    //Last Writer Stuff Mapping
85    for(int i = 1; i < N; i++){
86       for(int j = 1; j < N; j++){
87          std::vector<int> wref{i,j};
88          assert(shadow[i][j] == polyfunc::lastWriter(max_bound,
89          wref, wref_fix_flag, mapping.LHS_MAP));
90       }
91    }
92    //================================================================
93
94
95    return 0;
96 }
97
```

Listing 4: The main function of the hand instrumented code

4.1.2 Initializing shadow variables

Line 5-27 is about initializing shadow variables. The data space of a program, in the beginning, has two conditions. First, we either write to it at the beginning (like A[1][1] in listing 1) or we read from it at the beginning (like A[0][1] in listing 1). To implement this part, we assume that the data space is rectangular. Thus, we make this initialization by dividing the statement into its write reference and read references (like the original statement). The tricky part is to find the induction variables that affect the statement to generate the for loops. Here from the three variables of the statement instance $S < t, i, j >$, we only need "i" and j loops. This isn't a very scientific approach, but I should have simplified some assumptions to make the implementation manageable.

First, we extract the write references and read references affine functions in "initPolyCheckVariables" to initialize the shadow variable. Then, we make sure that the statement is invalid for the write reference and cannot be read from any other statement instance. For the data space that we will read from them, I will make them valid and put the init statement inside them. The init statement is for further initialization steps. The valid flag is a flag that detects many other situations that may generate invalid statement instances.

4.1.3 Mapping

Line 30-36 relates to the required mapping. To compute the mapping part, we need two things. First, a mapping from the data space to the related iteration space, e.g., in the listing 1 we need a mapping from write reference to the related iteration space (i,j). Then we also need a mapping from the iteration spaces to the reading space. To implement this part using ROSE, we assume that each subscript is in the form of a linear function of a * t + b * i + c * j + ... + d. Then we extract a, b, c ..., and d. Using these variables, we will drive the required mapping. For example for iterations space to first read reference A[i - 1][j], we simply need to dot product (0, 1, 0) with (t, i, j) + (-1) to drive the read reference of [i - 1].

4.1.4 min_bound, max_bound, wref_fix_flag, read_const

Variables defined in lines 39 to 60 are going to be used to input the functions of Polycheck. Their initialization is trivial and can be computed efficiently using the previous steps' implementations.

4.1.5 Test function

Line 61-82 relates to initializing the array and running the two functions, and comparing their results. It initializes the array with some random variable from uniform distribution and then runs two functions: Original and Transformed_Seidel, and tests their output. Just like a naive testing implementation that we are normally using to debug the code.

4.1.6 Original_Seidel function

This function is the input in this example. In this function, we are iterating over a 2D space. Then we are reading from two elements and writing to another element. Please see the figure 1. It shows the read and writes data spaces. The code is reading from the red elements (elements {1,4}) and write into the green one (element {5}). Note that there are 6 elements, {1,2,3,4,8,12} that we are only reading from them, and at the beginning, we are writing to elements {5,6,7,9,10,11,13,14,15}.

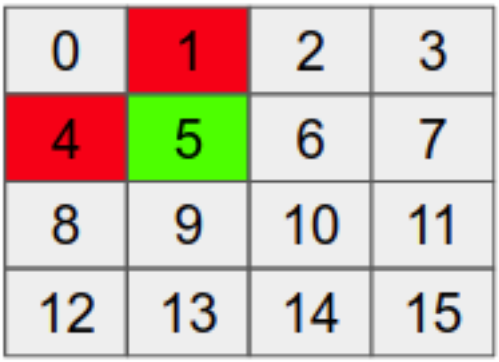

Figure 1: The read and write scheme in the input code

4.1.7 Transformed_Seidel function

The transformed function recursively tile the input elements until it gets to the tile size of one. In figure 2, I showed the tiling scheme. This is one example of non-affine transformation.

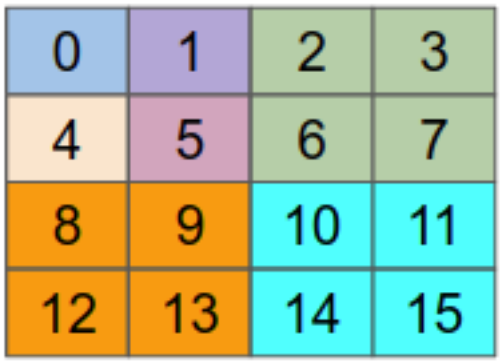

Figure 2: The recursive algorithm partitioning

4.1.8 LastWriter

Line 84-90 checks the last statement instance that write into a specific data space. The procedure is as follows. First, it will get the write reference that is for a data space A[i][j] the write reference (wref) is (i,j) (line 87). Using the LHS_MAP and wref_fix_flag, we drive the two inductive variables i and j in $S < t, i, j >$. Based on the original code in listing 1, the last statement instance that writes into this specific data space is t = T - 1. They do this computation using the ISL library, which computes these faster using a specific data structure in the backend. However, for these small tests, I don't think that there would be a meaningful difference in the overhead of these assertions. Please see the listing 5 for further implementation details. Line 16-20 is the mapping part, and lines 23 to 29 drive the last writer statement instance. After doing this computation for each data space, we can ensure that the program is finished completely.

```
1  StateInst lastWriter(const StateInst& max_b,const std::vector<int>& wref,
2  const std::vector<bool>& fix_flags,const std::vector<std::vector<int>>&
       mapping){
3     StateInst result;
4     result.instance.resize(fix_flags.size());
5
6
7     if(mapping.size() != wref.size()){
8     std::cerr << "The␣mapping␣dimension␣doesn't␣match␣the␣wref␣dimentions" <<
       std::endl;
9     }
10
11    if(std::count(fix_flags.begin(), fix_flags.end(), true) != wref.size()){
```

```
12    std::cerr << "The␣number␣of␣fixed␣iterations␣doesn't␣match␣the␣flag␣
      indicator" << std::endl;
13    }
14
15    //Compute the fixed iteration space
16    std::vector<int> fixed_iter_space(wref.size());
17    for(int row = 0; row < wref.size(); row++){
18       for(int col = 0; col < wref.size(); col++)
19          fixed_iter_space[col] += mapping[row][col] * wref[col];
20    }
21
22    int cnt = 0;
23    for(int i = 0; i < fix_flags.size(); i++){
24       if(!fix_flags[i]){
25          result.instance[i] = max_b.instance[i] - 1;
26       } else {
27          result.instance[i] = fixed_iter_space[cnt];
28          cnt++;
29       }
30    }
31
32    result.makeValid();
33    result.makeNoInit();
34
35    return result;
36 }
37
```

Listing 5: The lastWriter fucntion

4.2 Instrumented function

Now we need to instrument the Transformed-Seidel function for runtime checks. First, we need to find the statement instance that we want to instrument. Then we are inserting the required checks, using the variables that we have just defined and functions with SageInterface. For example, I am using SageInterface::addTextForUnparser function for most of my instrumentation. I also use SageInterface::insertStatementAfter occasionally. Here, I am more focused on explaining the details of the functions and their implementation rather than the compiler part since that part is more repetitive work.

We first drive the wref (Line 5). Note that we don't care how "ilo" and "jlo" are obtained, and we only care about the values of these two variables, and thus, I am using their string representation to define wref. Given the last statement instance written in A[ilo][jlo], which is store in shadow[ilo][jlo] we have two situations. It is either the first time we want to write in this data space, or it is one of the writes coming after a previous write into this data space. If it is the first time, we use firstWriter function (Line 8). The procedure of this function is

similar to the lastWriter function. However, instead of using the max_bound, we are using the min_bound, and we are going back in time.

```
void Instrumented_Seidel(int t,int ilo,int ihi, int jlo, int jhi){
if (ilo > ihi || jlo > jhi) return;
if (ilo == ihi && jlo == jhi){
 A[ilo][jlo] = A[ilo-1][jlo] + A[ilo][jlo-1];
 std::vector<int> wref{ilo, jlo};
 polyfunc::StateInst optStat;
 if(shadow[ilo][jlo].isInit() || !shadow[ilo][jlo].isValid()){
  optStat = polyfunc::firstWriter(min_bound, wref, wref_fix_flag, mapping.
    LHS_MAP);
  assert(optStat.isValid());
 } else {
  optStat = polyfunc::nextWriter(shadow[ilo][jlo], max_bound, wref,
    wref_fix_flag, mapping.LHS_MAP);
  assert(optStat.isValid());
 }

 //Checking the right hand side
 std::vector<int> read_ref{ilo - 1, jlo, ilo, jlo - 1};
 for(int rr_ptr = 0; rr_ptr < read_ref.size(); rr_ptr++){
 assert(polyfunc::dotProduct(mapping.RHS_MAP[rr_ptr], optStat.instance) +
   read_const[rr_ptr] == read_ref[rr_ptr]);
 }

 //Checking the validity of the value of rhs arrays
 std::vector<int> read_ref0(read_ref.begin(), read_ref.begin() + 2);
 assert(shadow[ilo - 1][jlo] == polyfunc::writeBeforeRead(optStat, min_bound
   , read_ref0, wref_fix_flag, mapping.LHS_MAP));
 std::vector<int> read_ref1(read_ref.begin() + 2, read_ref.begin() + 4);
 assert(shadow[ilo][jlo - 1] == polyfunc::writeBeforeRead(optStat, min_bound
   , read_ref1, wref_fix_flag, mapping.LHS_MAP));
 shadow[ilo][jlo] = optStat;
 //=========================================================


 }else{
    Instrumented_Seidel(t, ilo,(ilo + ihi) / 2, jlo, (jlo + jhi)/2); //Top-
   Left
    Instrumented_Seidel(t, ilo,(ilo + ihi) / 2, (jlo + jhi) / 2 + 1,jhi); //
   Top-Right
    Instrumented_Seidel(t, (ilo + ihi) / 2 + 1, ihi, jlo, (jlo + jhi) / 2);
   //Bottom-Left
    Instrumented_Seidel(t, (ilo + ihi) / 2 + 1, ihi, (jlo + jhi)/2 + 1, jhi)
```

```
    ; //Bottom-Right
35  }
36
37
38  if (ilo == 1 && ihi == N - 1 && jlo == 1 && jhi == N-1 && t < (T - 1)){
39    Instrumented_Seidel(t+1, ilo, ihi, jlo, jhi);// Next time step
40  }
41 }
42
```

Listing 6: The instrumented code

If it is the second situation, we should use nextWriter function (Line 11). This function aims to get the previous statement instance that writes into this place and generate the next writer using the original schedule of the original program. For example, after seeing the statement instance $S < 1, 5, 6 >$, using the original schedule should generate $S < 2, 5, 6 >$ as the following statement instance that writes into this place. Please note that these functions are general, and we need to use wref and mapping to find the write statement instance.

Please see listing 7. For simplicity, I removed some of the condition checks in the function. For the original implementation, please see the attached code. In lines 27-32, we will find the first inner loop that doesn't affect the wref. Then, we start by finding the first statement instance that writes into this specific place after the previous statement (line 34 to 42). If we find a valid statement instance, then we are good to go for the next step.

```
1  StateInst nextWriter(const StateInst& prev,const StateInst& max_b,const std
       ::vector<int>& wref,
2  const std::vector<bool>& fix_flags, const std::vector<std::vector<int>>&
       mapping){
3     StateInst result;
4     result.instance.resize(fix_flags.size());
5     result.makeNoInit();
6
7     std::vector<int> fixed_iter_space(wref.size());
8     for(int row = 0; row < wref.size(); row++){
9        for(int col = 0; col < wref.size(); col++)
10          fixed_iter_space[col] += mapping[row][col] * wref[col];
11    }
12
13    //Create the correct instance
14    int cnt = 0;
15    for(int i = 0; i < fix_flags.size(); i++){
16       if(!fix_flags[i]){
17          result.instance[i] = prev.instance[i];
18       } else {
19          result.instance[i] = fixed_iter_space[cnt];
20          cnt++;
```

```
21        }
22    }
23
24    //compute the nextwriter
25    //Compute the place where incrimination begins
26    int starting_point = 0;
27    for(int i = 0; i < result.instance.size(); i++){
28        if(fix_flags[i]){
29            starting_point = i - 1;
30            break;
31        }
32    }
33
34    for(int i = starting_point; i >= 0; i--){
35        int tmp = result.instance[i] + 1;
36        if(tmp < max_b.instance[i]){
37            result.instance[i] = tmp;
38            result.makeValid();
39            return result;
40    } else {
41        continue;
42    }
43
44    result.invalidate();
45    return result;
46 }
```

Listing 7: nextWriter function

After computing the statement instance, we know the correct statement that should write into this place right now. So, we have to check whether the current operation is equal to the computation of the statement instance that we just computed or not. In listing 6, line 16-19. Using the RHS_MAP drive from the original code, we are converting the iteration space of the statement instance that we drove into the read data space. Then we are checking these read data spaces with the read data space that is currently happening. If they are equal, we will go to the next step, checking the values of these data.

In lines 22-26, we are using the last vital function writeBeforeRead. This function can reflect the use of mapping better. Here, we use the statement instance we just computed as a time reference, and we want to drive a statement instance that writes into some data space that occurs before the current statement instance. For example, assume that we are currently computing $S < 1, 5, 7 >$. The last statement instance that writes into A[4][7] is $S < 1, 4, 7 >$, and the last one that writes into data space A[6][7] is $S < 0, 6, 7 >$. In listing 8 we first need to find the fixed inductive variables that always write into specific data space. Note that it is similar to nextWriter and lastWriter function. The only difference is that now we are working on reading references rather than wref. Please note that we still use the LHS_MAP since the mapping for that specific statement instance that writes into the

current read reference is the same. Also, please see lines 34-42 that show the going back in time.

```
1
2  StateInst writeBeforeRead(const StateInst& current_read, const StateInst&
       min_b, const std::vector<int>& wread,
3  const std::vector<bool>& fix_flags, const std::vector<std::vector<int>>&
       mapping){
4     assert(current_read >= min_b);
5     StateInst result;
6     result.makeNoInit();
7     result.instance.resize(fix_flags.size());
8
9     std::vector<int> fixed_iter_space(wread.size());
10    for(int row = 0; row < wread.size(); row++){
11       for(int col = 0; col < wread.size(); col++)
12          fixed_iter_space[col] += mapping[row][col] * wread[col];
13    }
14
15    //Create the correct instance
16    int cnt = 0;
17    for(int i = 0; i < fix_flags.size(); i++){
18       if(!fix_flags[i]){
19          result.instance[i] = current_read.instance[i];
20       } else {
21       result.instance[i] = fixed_iter_space[cnt];
22       cnt++;
23       }
24    }
25
26    if(result < current_read){
27       //It is a data form Input set that not yet being
28       // written by any other statement instance
29       for(int i = 0; i < result.instance.size(); i++){
30          if(result.instance[i] < min_b.instance[i]){
31             result.makeValid();
32             result.makeInit();
33             return result;
34          }
35       }
36
37       result.makeValid();
38       result.makeNoInit();
39       return result;
40    }
```

```
41
42      //Compute the place where decrement begins
43      int starting_point = 0;
44      for(int i = 0; i < result.instance.size(); i++){
45         if(fix_flags[i]){
46            starting_point = i - 1;
47            break;
48         }
49      }
50
51      for(int i = starting_point; i >= 0; i--){
52         int tmp = result.instance[i] - 1;
53         if(tmp > min_b.instance[i]){
54            result.instance[i] = tmp;
55         } else {
56         continue;
57         }
58      }
59
60      //It is a data form Input set that not yet being
61      // written by any other statement instance
62      for(int i = 0; i < result.instance.size(); i++){
63         if(result.instance[i] < min_b.instance[i]){
64            result.makeValid();
65            result.makeInit();
66            return result;
67         }
68      }
69
70      result.makeValid();
71      result.makeNoInit();
72      return result;
73  }
```

Listing 8: writeBeforeRead function

5 Implementation of the Original Schedule and print statement in Compiler

Although I am using a limited representation of the original schedule, and now a predefined function to print the statement and use these in the current implementation, but I have also implemented these function at the beginning from the scratch.

Please see the figure 3 and its corresponding code in figure 4. In here, we can see that the tree is in levels and each levels defined by the statements and the level of nesting

that they have. So, basically to extract the original schedule we need to drive these levels, and for each level, we will assign an statement instance shape. To do that, I am using "AstTopDownProcessing" to extract these information and for each SgExprStatement, we are adding the template using addNewAttribute function and attribute class to the nodes in the AST tree.

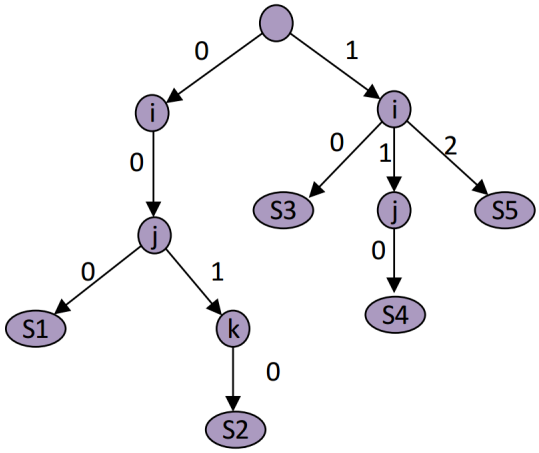

Figure 3: The original schedule graph that shows the dependency between statement instances in an affine program

```
for (i = 0; i < n - 1; i++){
  for (j = i + 1; j < n; j++){
    S1: f = a[j,i] / a[i,i];
    for(int k = i + 1; k < n; k++){
      S2: a[j,k] -= f * a[i,k]
    }
  }
}
for(i = 0; i < n; i++){
  S3: s = 0;
  for(j = n - i + 2; j < n; j++){
    S4: s += a[n - i + 1, n + 1];
  }
  S5: x[n - 1 + 1]++;
}
```

Figure 4: The toy example that we use to drive the original schedule of the program

For example, please consider statement S1. It is in the 2 nested for-loops. S1 shape is $S < 0, i, 0, j, 0 >$ where the last 0 is its priority inside the basic block of the two nested loops. So, any statement in this 2 nested for loops will have $S < 0, i, 0, j >$ in their beginning. Using

this information and the traversing method that I have mentioned, I have implemented a general original schedule retriever.

As for the printing function, I implemented it using a AstBottomUpProcessing traversal. Then, each node will gather the string from its child in the AST tree and add its own string definition to it and parse it up in the AST. Please see listing 9 for a detailed implementation.

```
1
2  std::string PrintSubscripts::evaluateSynthesizedAttribute(SgNode* node,
       SynthesizedAttributesList child_att){
3     if(isSgBinaryOp(node)){
4        if(isSgAddOp(node)){
5           return child_att.front() + "␣+␣" + child_att.back();
6        } else if(isSgSubtractOp(node)){
7           return child_att.front() + "␣-␣" + child_att.back();
8        } else if(isSgMultiplyOp(node)){
9           return child_att.front() + "␣*␣" + child_att.back();
10       } else if(isSgDivideOp(node)){
11          return child_att.front() + "␣/␣" + child_att.back();
12       }
13    } else if(isSgValueExp(node)){
14       if(isSgIntVal(node)){
15          return std::to_string(isSgIntVal(node)->get_value());
16       } else if(isSgDoubleVal(node)){
17          return std::to_string(isSgDoubleVal(node)->get_value());
18       } else if(isSgFloat128Val(node)){
19          return std::to_string(isSgFloat128Val(node)->get_value());
20       }
21    } else if(isSgVarRefExp(node)){
22       return isSgVarRefExp(node)->get_symbol()->get_name();
23    } else {
24       std::cerr << "This␣is␣not␣a␣valid␣subscript" << std::endl;
25    }
26    return "";
27 }
```

Listing 9: Buttom up traversal - Based on the type of the node we will retrieving its string and pass it to its parent.

6 A note about Algorithm B in the paper

If a statement instance can pass all of these tests, then it is a valid statement instance. There is one more algorithm in the paper. The terminology is the same. However, even with ISL, the overhead was huge, so they decided to work with numbers instead of integer sets. That is, for example, the statement instances that write into A[5][7] (like S¡0,5,7¿, S¡1,5,7¿, etc.) can be mapped into a sequence of numbers. They present some conditions that guarantee

that this mapping exists, and then they use these sequences of numbers to check all those assertions.

7 Evaluation

To test for this example, I change the boundaries of the recursive calls to see whether the Polycheck demo can detect the errors. It successfully detects these bugs while the test suit failed to generate errors.

Unfortunately, as you have noticed, this implementation doesn't cover many corner cases. Since it is not integrated with the PoCC compiler, I could not check it with a benchmark, and I am sure that it will work with this example. As a hands-on experiment, I am happy that I could understand a small compiler research space and work with one scientific compiler. However, this implementation by no means is complete, and it cannot extends to general cases. But I did my best to grab the core idea of the PolyCheck paper.

As for the overhead, with T=3 and N=64 in the Seidel example, please see the runtime:

- Time taken by Original_Seidel: 8 microseconds
- Time taken by Transformed_Seidel: 64 microseconds
- Time taken by Instrumented_Seidel: 1548 microseconds

As we can see, the overhead is not that small! That is why they come up with algorithm B at the end. But, I believe that the bottleneck is also a direct cause of the use of the shadow variable itself rather than the computations alone.

To compile the code on Ubuntu, install ROSE using the official repo they have provided using the apt install command (it will not take a lot of time compared to compiling the rose from the source). Then please change the CMakeLists.txt based on the address of the library in your machine. If you want to see a demo, please go to the demo folder and use the "Project" executor with the address of your input code (Though I think it will not work because of some dynamic libraries that I have used). The rose_* will be the instrumented code. Note that you should have a Transformed_* function and an Original_code* for instrumentation purposes in the input code. Please note that the header file polycheck_demo_functions.h should also be included since the instrumentation uses the function in this header file.

8 Discussion

As we discussed in the project presentation, the overhead of this implementation is enormous, and we cannot detect bugs until they happen!. The data also needs to move inside the same control flow, and if there are branches that depend on specific data, the verification will still be data-dependent. However, it is now more independent from the data than other testing technics that rely on the data results. It is a good technique for debugging since it can also approximately detect where the error is happening. Still, it cannot be used in sensitive works because of its overhead and that it will detect errors in the late stages.

Please note that the overhead of this technique is still much less than the previous trace-based work, namely CodeThorn.

9 Summary

References

[1] M. Schordan, P.-H. Lin, D. Quinlan, and L.-N. Pouchet, "Verification of polyhedral optimizations with constant loop bounds in finite state space computations," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pp. 493–508, Springer, 2014.

[2] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan, "Polycheck: Dynamic verification of iteration space transformations on affine programs," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 539–554, 2016.

[3] V. Loechner, "Polylib: A library for manipulating parameterized polyhedra," 1999.

[4] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software (ICMS'10)* (K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, eds.), LNCS 6327, pp. 299–302, Springer-Verlag, 2010.

[5] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel processing letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[6] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International journal of parallel programming*, vol. 21, no. 6, pp. 389–420, 1992.