Using Dependence Analysis and Loop Transformations to Promote Parallelism in Java.

Eugene Bulog, Nathan Cairns and Kerwin Sun
Department of Electrical, Computer, and Software Engineering
University of Auckland, Auckland, New Zealand
ebul920, ncai762, ksun182

***Abstract* - Parallelization is vital to creating quick and efficient programs. Loops are very attractive code blocks to parallelise as they are where most computation is done. Dependence analysis detects whether dependencies are present and hence whether a block of code is parallelizable. Loop transformations can be used to make a loop more parallelizable by removing dependencies. In this paper we explore dependence analysis, loop transformations and present a Java code transformer which uses the polyhedral approach.**

***Keywords* - Loop Transformations, Dependence Analysis, Parallelisation, Optimisation**

I. Introduction

Parallelisation is the act of allowing sequential code to be executed in parallel. This means that the code will be broken up to be executed across several processors or threads. One of the most common structures we see in code is loops. Due to the commonality of loops and the potential for loops with large amounts of iterations, we are provided with a rich opportunity to speed exploit parallelism and speed up our code [1]. Whether a block of code can be parallelised is determined by the existence of data dependence [2]. This means any part of the program in which one thread consumes data read by another thread [3]. This can be especially prevalent in loops where one iteration can access data modified in a previous iteration, stopping us from breaking up iterations across processors/threads. Hence, a very popular compiler operation is detecting data dependencies to determine which code can be parallelised and which cannot. Once a dependency is detected we can apply several code transformation techniques to attempt to eliminate these dependencies and make the source code more parallelizable.

In this report we provide a summary of the concepts from the relevant literature on dependence analysis and loop transformations. This includes the types of dependencies, the current dependence analysis approaches, the polyhedral approach to dependence analysis, and some popular loop transformation methods. Following this we describe a system we have implemented to demonstrate the polyhedral approach on for loops written in Java. Finally, we discuss the successfulness of our approach and the future work which could be pursued.

II. Relevant Literature

A. *Types of Dependencies*

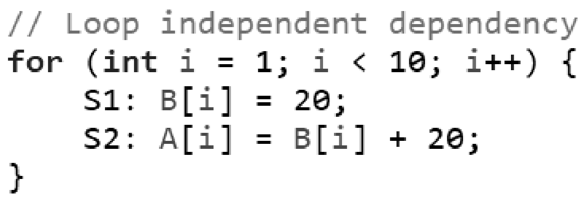
Before we explore dependence analysis it is first important to learn what we are looking for by learning the types of dependencies which exist. There are two overarching types of dependencies, these are control dependence and data dependence. Control dependence is any statement which decides what statement executes next. For example, an if statement. Since control dependencies can be written as data dependence and data dependence decide whether code can be parallelised, we will focus primarily on data dependence. There are four types of data dependence. These are [4]:

- **Flow Dependence** is when data is written by one instruction and then read by another.
- **Anti-Dependence** is the opposite of flow dependence, it is when data is read by one instruction and then read by another.
- **Output Dependence** is when data is written to by one instruction and then written to again by another.

- **Input Dependence** is when data is read from by one instruction and then read from again by another. Input dependence is generally not considered an obstacle to parallelism.

Data dependencies can appear in two different ways in loops. These are as follows [2]:

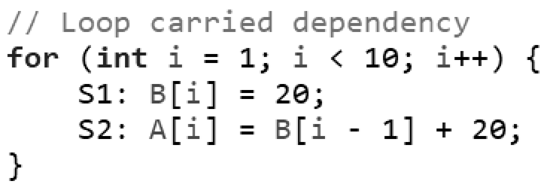
- **Loop Independent Dependencies**, these are intra-dependendent loop dependencies, i.e. dependencies which appear between statements on the same iteration of a loop. They are generally handled by the privatisation of variables. An example of a loop independent dependency can be seen in Figure 1. In this case S2 is flow dependent on S1 in the same loop iteration. Hence, a loop independent dependency exists.



```
// Loop independent dependency
for (int i = 1; i < 10; i++) {
     S1: B[i] = 20;
     S2: A[i] = B[i] + 20;
}
```

Figure 1: Loop independent dependency example

- **Loop Independent Dependencies**, these are intra-dependendent loop dependencies, i.e. dependencies which appear between statements on the same iteration of a loop. They are generally handled by the privatisation of variables. An example of a loop independent dependency can be seen in Figure 2. In this case S2 is flow dependent on S1 in the same loop iteration. Hence, a loop independent dependency exists.



```
// Loop carried dependency
for (int i = 1; i < 10; i++) {
    S1: B[i] = 20;
    S2: A[i] = B[i - 1] + 20;
}
```

Figure 2: Loop carried dependency example

B. *Dependence Analysis*

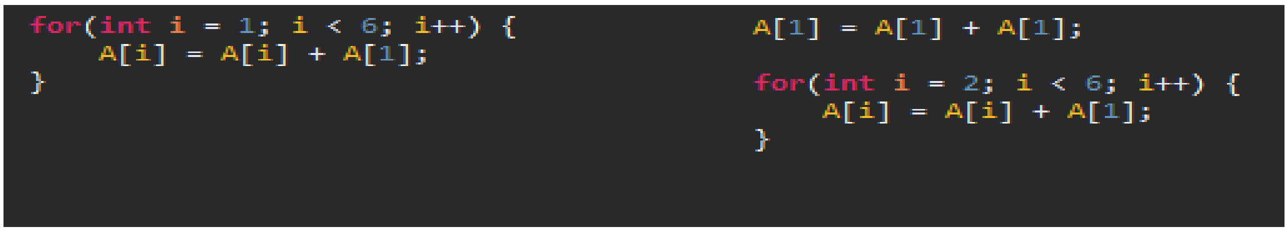
Data dependencies between iterations of a loop impose restrictions in the order of execution of statements and hence restrict the degree to which a loop can be parallelised. Compilers hence rely on data dependence analysis tests for finding dependencies and determining whether code can be parallelised [5]. Since these tests generally happen at compile time there is a tradeoff between efficiency and accuracy. This is so a compiler does not end up having unreasonably long compile times. This sacrifice of accuracy means data dependence tests always approximate on the conservative side. I.e. a dependence is assumed if independence cannot be proved [6], ensuring no unsafe parallel programs are produced. Most of these tests require loop bounds and array subscripts to be represented as a linear (affine) function of the loop index variables [8]. The presence of dependencies can then be checked by finding an integer solution to these functions which satisfy a set of linear inequality constraints.

There are several dependency tests these include, the GCD test, the Banerjee test, the I-Test, the Omega Test [6], and more. The GCD test is the simplest dependency test and provides a good example of how one of these tests work. It works by computing the GCD (Greatest Common Divisor) of the coefficients in the left hand side of the equation. If this GCD evenly divides the constant on the right-hand side of the equation there might be a dependency, otherwise there is not. Hence it is a necessary but not sufficient condition for dependence [11].

C. *Loop Transformations*

There are many varieties of loop transformations possible, and this section will briefly outline several of the better-known ones.

Loop splitting is a loop transformation that can be used when many iterations all depend on a single iteration or chunk of iterations. This transformation involves separating dependent and non-dependent iterations into separate loops, meaning that iterations of one loop can be parallelized once the iterations they depend on have been separated out. Loop "peeling" is a specific subtype of splitting, wherein the first iteration is removed, if subsequent iterations all depend on specifically the first one, as shown on the left side of Figure 3. This single "problematic" iteration can be "peeled" out of the loop and placed in a separate loop or as a statement outside of the loop (as shown on the right side of Figure 3, to be executed sequentially, after which the remainder of the loop's iterations can be run in parallel with each other [2].



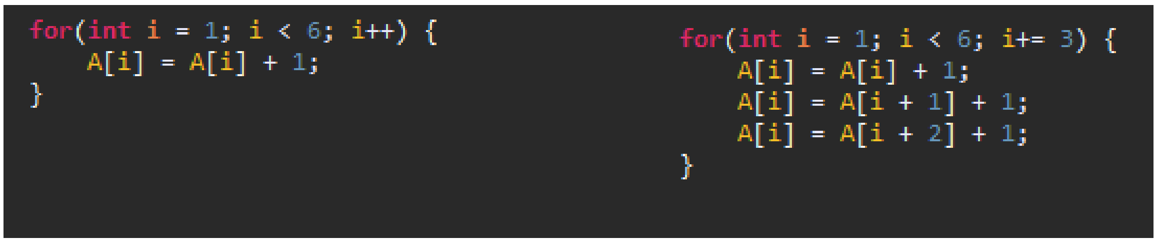
```
for(int i = 1; i < 6; i++) {
    A[i] = A[i] + A[1];
}
```

```
A[1] = A[1] + A[1];

for(int i = 2; i < 6; i++) {
    A[i] = A[i] + A[1];
}
```

Figure 3: An example of a loop before (left) and after (right) peeling

Loop unrolling is the action of combining multiple loop iteration bodies into a single body, and then increasing the rate at which the loop counter increments, as shown in Figure 4. This results in less granular parallelization (when iterations are parallelized), as well as reducing the overhead of checking the loop condition after each iteration (as there are now less iterations).

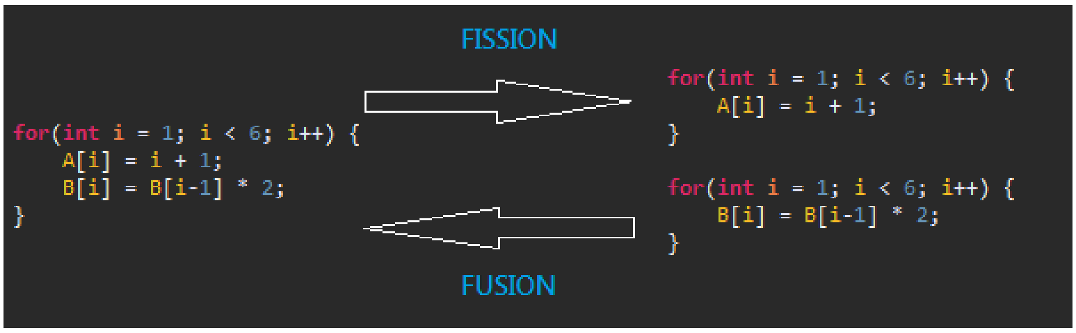


```
for(int i = 1; i < 6; i++) {
    A[i] = A[i] + 1;
}
```

```
for(int i = 1; i < 6; i+= 3) {
    A[i] = A[i] + 1;
    A[i] = A[i + 1] + 1;
    A[i] = A[i + 2] + 1;
}
```

Figure 4: An example of a loop before (left) and after (right) unrolling

Loop fission is the process of splitting a single loop containing multiple body statements into multiple loops (with the same conditions but different body statements) to change the granularity of parallelism, or by splitting non-dependent and dependent statements into separate loops, allowing, for example, one loop to be parallelized when previously the original loop could not be, as shown in Figure 5. Loop fusion is the reverse of this process to increase granularity or reduce loop overhead [2].



```
for(int i = 1; i < 6; i++) {
    A[i] = i + 1;
    B[i] = B[i-1] * 2;
}
```

```
for(int i = 1; i < 6; i++) {
    A[i] = i + 1;
}

for(int i = 1; i < 6; i++) {
    B[i] = B[i-1] * 2;
}
```

Figure 5: An example of loop fission and fusion

Another common transformation that can be used to optimize loops is loop interchange. Interchange is a transformation that can be used when the order of access of array members may not be the same as the order in which members are stored, depending on the memory architecture of the runtime machine. For example, if a machine has column-major array storage, then parallelizing in such a way that members are accessed in row-major fashion by the same thread can lead to inefficiencies and a higher chance of cache misses. Interchange solves this by “switching” the inner and outer loop variables, in order to change from column-major access for inner iterations to row-major, or vice versa. While loop interchange is useful for optimizing parallelization, it does not generally have much use for making “unparallelizable” loops more easily parallelizable, and instead benefits performance [2]. Figure 6 illustrates and compares the two forms of array ordering.

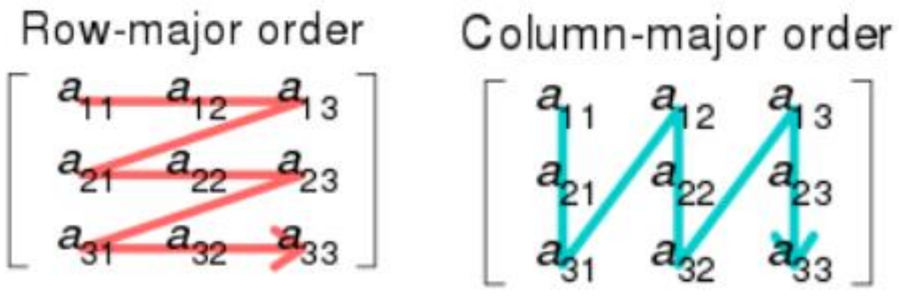
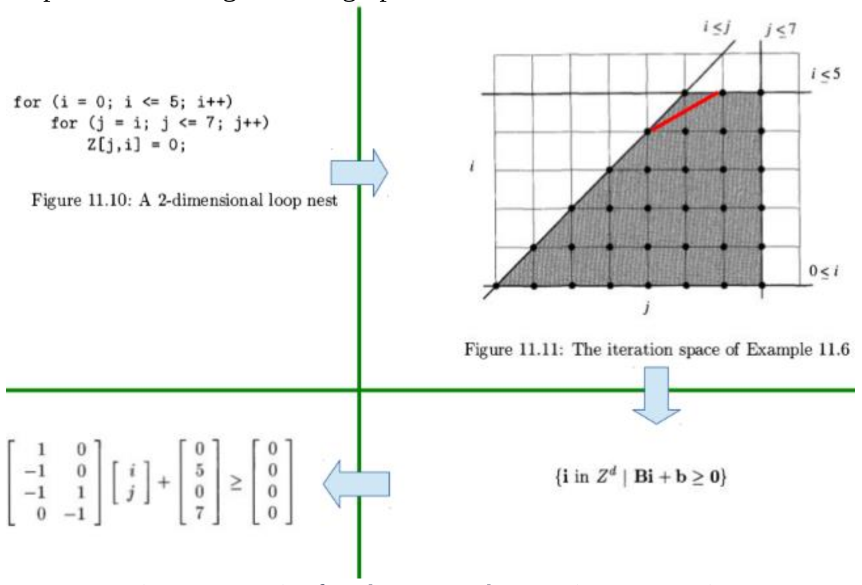

Row-major order
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Figure 6: A comparison of row-major and column-major array ordering

Most compilers (including the Java compiler) perform several of these loop transformations at compile-time, however, this step is often aimed at optimizing runtime performance, rather than affecting parallelism, which is the goal of this project. Furthermore, as this is done at compile-time, the developer does not have access to post-transformation source code, in cases when they wish to parallelize further once the loop has been transformed into a form that supports such parallelization.

D. Ployhedral Approach

The polyhedral approache aims to model Loops in an X dimensional projection/graph where the components/axis of the graph are the components of the loop and each node is one iteration of the loop [9]. This can then be represented as a matrix inequality equation form ax + b => 0, where rows in the matrix define the bounds of the loop. Figure 7 demonstrates a mapping from Loop to Graph to Matrix representation. Dependencies of the loop between iterations are represented as edges in the graph from one node to another.



```
for (i = 0; i <= 5; i++)
    for (j = i; j <= 7; j++)
        Z[j,i] = 0;
```

Figure 11.10: A 2-dimensional loop nest

Figure 11.11: The iteration space of Example 11.6

$\{\mathbf{i} \text{ in } Z^d \mid \mathbf{Bi} + \mathbf{b} \ge \mathbf{0}\}$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ 0 \\ 7 \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 7: Mapping from loop to graph to matrix representation

Not all loops have a polyhedral representation, only Loops where iterations are in a convex set (any 2 points can be connected via a line without leaving the system) as shown in Figure 8. Representing a Loop in polyhedral form allows it to be bound by strict mathematical constraints, meaning regular transformations can be applied via computational means.

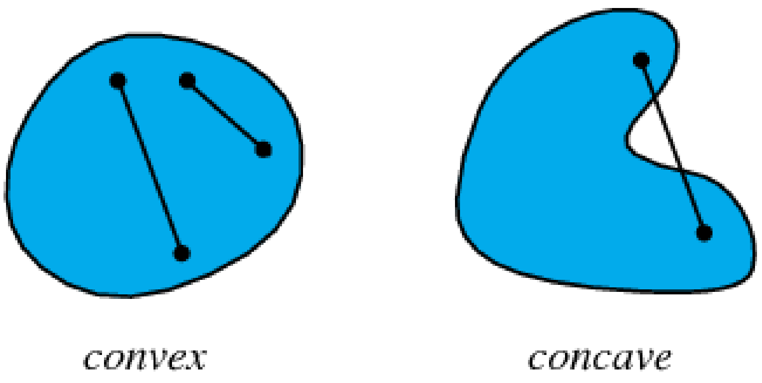


Figure 8: Convex and concave loop diagrams

E. Polyhedral Loop Transformations

Several transformations can be applied to polyhedral representations as a means of shifting or removing dependencies to allow for parallelization. These transformations must be affine meaning convexity and parallelism between point, lines and planes must be preserved pre and post transformation. Loop skewing and loop unrolling are considered affine as opposed to loop peeling/splitting where 'chunks' of the loop are displaced. Loop skewing involves remapping the components of a loop to remove dependencies along one component, by applying a transformation matrix t, such that the loop can be parallelized along that component. The inverse matrix t' gives us the mapping of the variables in the new representation [7]. Figure 9 presents a visual demonstration of loop skewing.

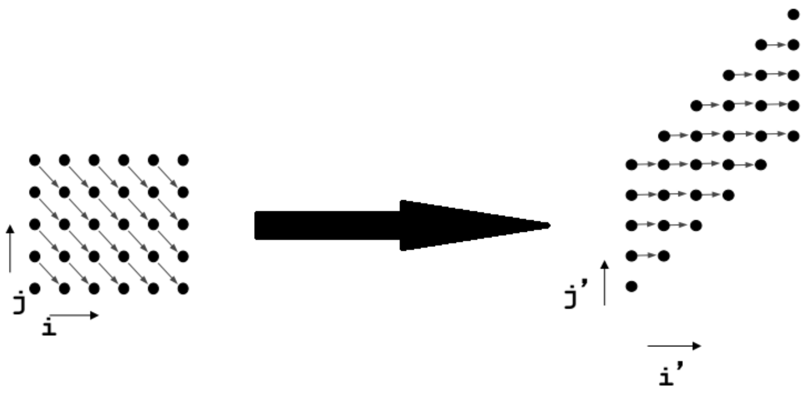


Figure 9: Loop skewing

III. IMPLEMENTATION – THE POLYHEDRAL APPROACH ON JAVA LOOPS

Our implementation 'Loopt' provides automatic loop skewing for 2 dimensional loops in java. Skewing is currently the only transformation Loopt supports. Loopt reads from a text input, automatically detects dependencies, runs transformation tests and returns a parallelizable, functionally equivalent loop snippet of the input if a viable skewing transformation exists. The architecture of Loopt can be seen in Figure 10 below. Implementation, documentation and tests can be found on our Github repository: https://github.com/Nathan-Cairns/Loopt

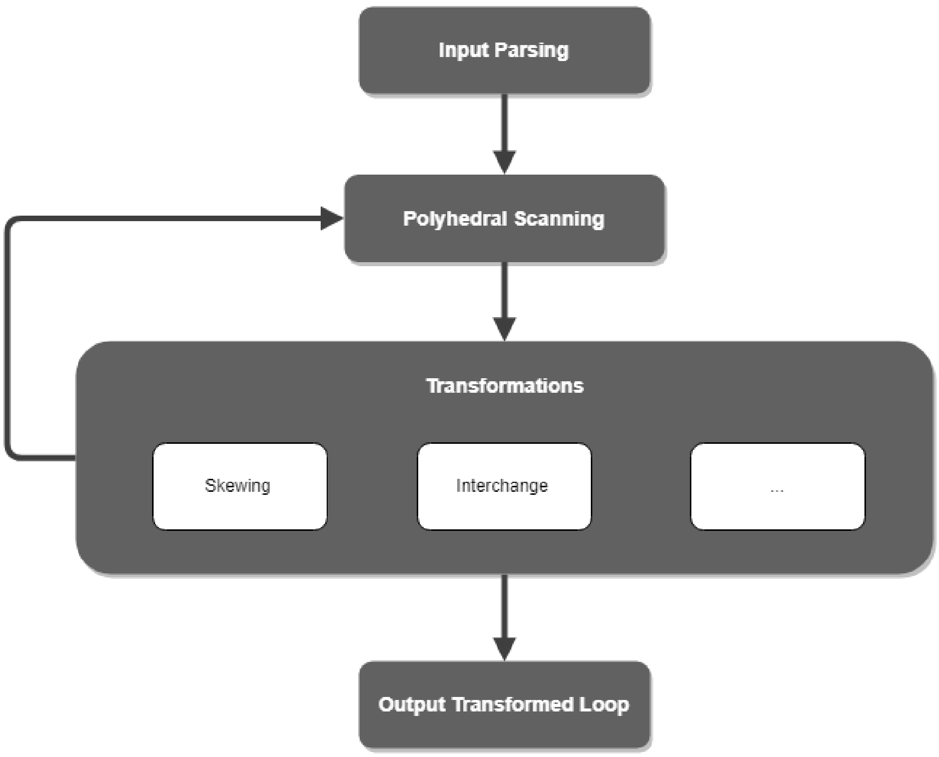


Figure 10: Loopt architecture diagram

A. Input Parsing

Loopt is interfaced with using a CLI (Command Line Interface). The CLI takes a java loop as input as either a string or file. The type of input is specified using a command line flag. Once the loop is parsed the dependency vectors and loop bounds are extracted. Our present implementation uses the open source Spoon library to do this. Once we have these dependency vectors and loop bounds we can perform affine transformations on the original loop code. This is discussed in the following section. It is important to note that there are several restrictions on the variants of loops which Loopt can currently analyse. These are as follows:

- Loops must be two dimensional (double nested)
- Loops must be for loops, with the following format: for (int i = x; i < n; i++)
- All array accesses must be at most as complex as simple arithmetic. Some valid examples are as follows:
  - A[i][j]
  - A[i + 1][j - 1]
  - A[i + 8][1]
- 2-dimensional dependencies cannot have any members that equal zero (e.g. a dependency vector of (1,0), i.e. both members must be non-zero integers).

Also, of note, is that due to limitations of the Spoon library used for parsing, input code snippets must include initializations (definitions are not required) for all variables being used in the snippet. The type the variables are initialized to is irrelevant. For example, if a snippet contains the array access A[i][j], the snippet must also contain an initialization for A, even if it is as simple as “int[][] A;”. This initialization does not actually affect the generated code at all (and is excluded from the output so long as it occurs outside the outer loop) but is simply required by Spoon to parse the input.

Once the Java code has been parsed and converted to an AST (Abstract Syntax Tree) by the Spoon library, dependencies are scanned for and represented as vector objects, which are passed to the transformation module.

B. Transformation Process

Once we were able to decompose loops into variable objects - LoopVars -specifying bounds and layer of each variable, it was easy to generate a 2-dimensional matrix polyhedral representation of the loop in the form Ax = B (Figure 7). Another key component was the dependency vector(D) extracted from the array access in the body of the loop, this was represented as an array list of DependencyVector objects, which contains a HashMap where the keys are the variables of the loop and the values are the linear dependency for that variable.

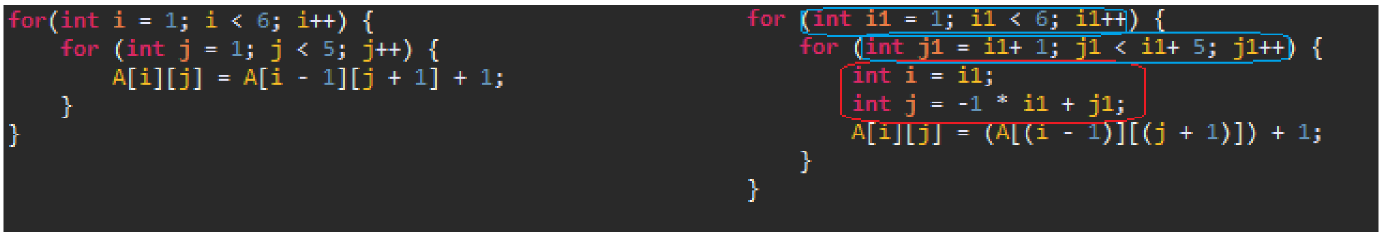
The challenge is now to generate a transformation matrix such that when applied to the dependency vector (t*D) one component of D is 0, meaning there is no dependency across that component. The determinant of t must also be 1 as the inverse of t, t' when applied to x should not give decimal values. Our system handles these 2 constraints by using a 'common multiples' brute force test method, for any dependency vector say d = (x, y), the system generates a list of common multiples such that xu + yw = 0 or xu = yw, for some arbitrary values of u and w. This constraint allows for satisfaction of the first constraint where the common multiple allows for cancellation of the vectors resulting in a 0 value in one component.

Next, for each common multiple 'M' found, the system generates the corresponding transformation matrix based on the the common multiple. The bottom row of the 2x2 [a,b; c,d] transformation matrix is set such that c = M/x and d = -(M/y), then values are tested between a predefined range for a such that (a*d - 1)%c == 0 and b = 0, this checks if there exists a for a and c value such that the determinant of t can be 1. If values for a and c are found, matrix t is returned, subsequently t' (inverse of t) is generated which gives us the variable mapping. It is important to note, the predefined testing range can be broadened to increase the likelihood of a valid transformation matrix hit, although this would increase the algorithms run-time it. This variable mapping allows us to replace the old bounds and array accesses of the loop with the new bounds.

This new loop will be parallelizable. In the case that the original loop had multiple array accesses resulting in multiple dependency vectors then the transformation matrix t is checked against the entire set. If the transformation works for all the members of the set then t is accepted, otherwise t is rejected, and the brute force algorithm continues.
EMLJ is the library used for Matrix representation and manipulation in Java, EMLJ offers a 'matlab like' interface for making procedural calls.

C. *Output Parsing*

Once the new post-transformation variable mappings have been generated, these variables as well as the original code snippet are passed to the output parsing module. The current implemented output module uses the Spoon library to rewrite the control statements for each loop (and nested loop), and to insert new definitions for the original loop variables based on the rewritten ones. As shown in Figure 11, this means that the bulk of the code stays the same, with only i and j (or equivalent loop variables) in the loop control statements being replaced with new variables (such as i1 and j1), the loop conditions being adjusted to match (circled in blue in Figure 11), and i and j being redefined in terms of i1 and j1 (circled in red in Figure 11). This approach is more flexible and easier to interpret than changing the code for every single affected variable access, as in this case a maximum of 2n lines needs to change, where n is the number of loops in the nest.



```
for(int i = 1; i < 6; i++) {
    for (int j = 1; j < 5; j++) {
        A[i][j] = A[i - 1][j + 1] + 1;
    }
}
```

```
for (int i1 = 1; i1 < 6; i1++) {
    for (int j1 = i1+ 1; j1 < i1+ 5; j1++) {
        int i = i1;
        int j = -1 * i1 + j1;
        A[i][j] = (A[(i - 1)][(j + 1)]) + 1;
    }
}
```

Figure 11: Input and output example for Loopt

D. *Testing*

As this project is not intended to be used at compile time or in any other time or performance-sensitive scenario, time and performance benchmarking are unnecessary and have been excluded. Of more importance, is the accuracy and limitations of the transformations performed. As part of this project, we have created a small test suite which reads ten pre-prepared .txt files containing snippets of Java loop code. Transformations of these snippets is attempted, and the transformed variable definitions are compared to the expected results (found by manually skewing the loops and adjusting to the format produced by our transformation module). This gives us a good idea of which types of dependencies and scenarios our system can handle. At the time of writing, 7 of the 10 test cases are correctly generating output, with the 3 remaining cases related to current limitations of the system (i.e. transforming loops containing multiple array accesses in one statement). This has allowed us to assess areas of future work that are currently outside of the scope of the project.

IV. Future work

As mentioned earlier, the current system can only perform skewing transformations. In theory, the polyhedral approach is versatile enough to allow many other types of transformations, so a future goal of this project is the addition of other transformations, as well as an automated method of analysing which transformation(s) is appropriate for a given snippet. Also, the current system is somewhat limited in the format of loops and types of dependencies that it can transform (see inputs section). This is an area that could also be improved in future, to make the system more versatile and robust.

V. Conclusion

Based on our research, there are multiple approaches possible for implementing a tool capable of transforming Java for loop code snippets. These possible approaches differ in the methods used to detect and analyze dependencies, as well as the transformations applied. For our implementation, we settled on using the polyhedral/polytope model for both dependency detection (via polyhedral scanning) and (affine) transformation of the loops. This approach was chosen for its flexibility and extensibility, as the polyhedral model can represent any affine loop space. Also, our implementation focuses solely on the skewing transformation, as skewing provides a good proof-of-concept transformation for demonstrating the power of the polyhedral model. The resulting system, although limited in the input it can take, successfully and dynamically applies the polyhedral approach to generate accurate and functionally equivalent transformed code in a variety of situations.

References

[1] Lilja, D. (1994). Exploiting the parallelism available in loops. Computer, 27(2), pp.13-26.
[2] Prema, S., Jehadeesan, R., Panigrahi, B. and Satya Murty, S. (2015). Dependency analysis and loop transformation characteristics of auto-parallelizers. 2015 National Conference on Parallel Computing Technologies (PARCOMPTECH).
[3] Midkiff, S. (2012). Automatic Parallelization: An Overview of Fundamental Compiler Techniques. Synthesis Lectures on Computer Architecture, 7(1), pp.1-169.
[4] Watkinson, N., Shivam, A., Chen, Z., Veidenbaum, A. and Nicolau, A. (2017). Using Data Dependence Analysis and Loop Transformations to Teach Vectorization. 2017 International Conference on Computational Science and Computational Intelligence (CSCI).
[5] Birch, J. and Psarris, K. (2008). Discovering Maximum Parallelization Using Advanced Data Dependence Analysis. 2008 10th IEEE International Conference on High Performance Computing and Communications.
[6] Psarris, K. and Kyriakopoulos, K. (n.d.). Data dependence testing in practice. 1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425).
[7] U. Banerjee, D. Gelernter and A. Nicolau, Languages and Compilers for Parallel Computing. New York: Springer, 1994.
[8] Blume, W. and Eigenmann, R. (1998). Nonlinear and symbolic data dependence testing. IEEE Transactions on Parallel and Distributed Systems, 9(12), pp.1180-1194.
[9] O. Dragomir and K. Bertels, "K-loops: Loop skewing for Reconfigurable Architectures", 2009 International Conference on Field-Programmable Technology, 2009. Available: 10.1109/fpt.2009.5377656 [Accessed 1 June 2019].
[10] "Efficient Java Matrix Library", Ejml.org, 2019. [Online]. Available: http://ejml.org/wiki/index.php?title=Main_Page. [Accessed: 03- Jun- 2019].
[11] Psarris, K. and Pande, S. (n.d.). Classical dependence analysis techniques: sufficiently accurate in practice. Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences.

Appendix

Appendix A: Table of Contributions

	Research	**Implementation**	**Presentation**	**Overall**
Nathan	33.3%	33.3%	33.3%	33.3%
Kerwin	33.3%	33.3%	33.3%	33.3%
Eugene	33.3%	33.3%	33.3%	33.3%