A Study of Transitive Closure based Loop Transformation Techniques of Affine Integer Relations for Coarse Grain Parallelism

Kiran Bhos

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौद्योगिकी संस्थान हैदराबाद

Indian Institute of Technology Hyderabad

Department of Computer Engineering

June 2015

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

[illegible]

(Signature)

(Kiran Bhos)

CS13M1011

(Roll No.)

Approval Sheet

This Thesis entitled A Study of Transitive Closure based Loop Transformation Techniques of Affine Integer Relations for Coarse Grain Parallelism by Kiran Bhos is approved for the degree of Master of Technology from IIT Hyderabad

[illegible]

(M. V. Panduranga Rao) Examiner
Dept. of Computer Science and Engg.
IITH

[illegible]

(Kotaro Kataoka) Examiner
Dept. of Computer Science and Engg.
IITH

[illegible]

(Dr. Ramakrishna Upadrasta) Adviser
Dept. of Computer Science and Engg.
IITH

[illegible]

(T. Bheemarjuna Reddy) Chairman
Dept. of Computer Science and Engg.
IITH

Acknowledgements

I am deeply indebted to my advisor Dr. Ramakrishna Upadrasta for his stimulating motivation and support. It was lot of learning experience working under him. I would also like to thank my whole class. I specially thank my parents, brother and sister for their constant support.

Dedication

To my parents, brother and sister

Abstract

This thesis focuses on computation of transitive closure of affine integer tuple relations and its effect on improvement on runtime of the resultant parallelized programs. Scalability issues of the computation are also discussed.
Different strategies are used by automatic parallelization compilers to find statements that can be executed in parallel. Most of the current approaches like Pluto [1] and Polly [2] use linear/integer-linear programming based techniques as a means to do the same. An emerging alternative is to use the transitive closure to do the same. The transitive closure based methods are different in strategy and complexity with compared with methods that use the linear programming based approaches. Traco [3] is a source to source transformation tool which tries to find slices of program that can be executed in parallel using Transitive Closure. Polly [2] is a branch of LLVM which uses scanning of AST to obtain independent dimension of iteration vector. Both Traco and Polly use OpenMP [4] pragmas to show detected parallelism. We do a comparative study of Traco and Polly to extract coarse grained parallelization. We suggest important modifications to Polly's algorithm of dependence extraction. We show limitations of the Traco compiler on various fronts: limitations in extracting parallelism, scalability because of dependence on transitive closure etc.

Contents

Declaration ii
Approval Sheet iii
Acknowledgements iv
Abstract vi

Nomenclature **viii**

1 Introduction **1**
1.1 What is Transitive closure 1
1.2 Who uses Transitive closure computation 2
1.3 Related work 3
1.4 Our contribution 3
1.5 Organisation of thesis 3

2 Background and Definitions **4**
2.1 Polyhedral Compilation 4
2.2 Affine Control Loop(ACL) 5
2.3 Mathematical Definitions 6
2.3.1 Rational Polyhedron vs Polyhedron 6
2.3.2 Lattice 6
2.3.3 Z-Polyhedra 6
2.3.4 Affine Transformation 7
2.3.5 Quasi-affine Integer Tuple Sets and Relations 7
2.3.6 Powers of Relation 8
2.3.7 Transitive Closures 8
2.3.8 Approximation 8

3 Literature Survey **10**
3.1 Motivating Applications 10
3.1.1 Iteration Space Slicing 10
3.1.2 Equivalence Checking 10
3.1.3 Free Scheduling 11
3.2 Current Work 11
3.2.1 Kelly et al. 11
3.2.2 Bielecki et al. 12

3.2.3 Verdoolaege, Albert Cohen et al. 12

4 Compilers for Coarse Grained Parallelization 14

4.1 Traco compiler 14

4.2 Optimization 14

4.3 Integration of Traco's functionality in Polly 15

4.3.1 Low Level Virtual Machine(LLVM) 15

4.3.2 Polly 16

4.3.3 Current working and Improvements proposed 16

5 Experiments with Two Polyhedral Compilers 17

5.1 Traco 17

5.1.1 Runtime Experiments 18

5.1.2 Compile Time Experiments 21

5.2 Polly 22

6 Conclusions and Future work 24

References 25

Chapter 1

Introduction

Now a days it is difficult to find someone who uses single-core machine. If we are using multi-core machines and running code in sequential manner, then that is a wastage of resources. To detect available parallelism in program is difficult task to do. It has always been a difficult task to manually analyze and detect parallelism in program. The task even become more complicated in auto-parallelism. There are some frameworks like OpenMP where we can manually annotate the parallelism in program and make best use of underlying multi-core hardware. But if we can do this automatically then that could save lots of time and efforts of programmer.

Amadahl's law states that, if f is a fraction of code parallelized, and if we are using p processors then speed up achieved is given by

$$\frac{1}{(1-f)+(f/p)}$$

So, if half of the computations are sequential then speed up can be at max doubled, regardless of number of processors. Most of time of program execution is passed during execution of loops. Parallelizing compilers tries to separate these iterations and execute them on different processors. A dependency analysis pass is performed on code to check if loop can be executed in parallel safely. And here computation of transitive closure comes in picture. Transitive closure is a technique that groups up all statement instances that are dependent on each other.

1.1 What is Transitive closure

For program analysis purpose, for example in dependence analysis, we cannot construct graph of all variables and their dependences. This is because the number of actual dependence instances are unbounded at compile time. It is generally not possible to enumerate all the related pairs of relation and then compute transitive closure. So parameterized integer tuple relations are used to summarize the dependence information. For example

```
for(i=3;i<=n;i++)
        a[i]=f(a[i-3]);
```

The dependence relation of above ACL can be written as.

$$R := \{[i] \to [i+3] \mid 1 \le i \le n\}$$

Transitive Closure for above relation can be written as follow.

$$\{[i] \to [j] \mid \exists k,\ i - j = 3k \wedge 3 \le i, j \le n \wedge (i > j \vee i < j)\}$$

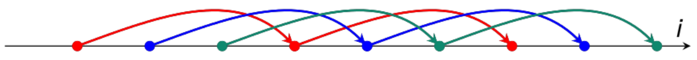

Figure 1.1: Example of Slicing

image source: [5]

When we apply standard Codegen functionality of Omega/Cloog, we get following equivalent code snippet. Here outer loop can be executed in parallel without being interfered by other slices.

```
for (i = 1; i <= min(n-3,3); ++i) { // parallel loop
        a[i] = f(a[i-3]);
        for (j = i + 3; j <= n; j += 3)
                a[j] = f(a[j-3]);
}
```

Here, even if we know value of parameter n we still we can't enumerate all vertices just for analysis purpose. Here what we are describing is family of infinite relations.

Linear programming based approaches takes cubic to fifth power of number of statements to find dependences while Transitive closure based approaches takes exponential time. Transitive closure approaches suffer from scalability issues but they give better results than linear programming based approaches.

1.2 Who uses Transitive closure computation

Computation of transitive closure is at heart of many applications. It is used in analysis of counter systems to accelerate the computation of counter systems. In counter systems the power of relation is used as "counting acceleration"[6]. In loop invariants computation, function bodies are treated as Transitive closure which applied on invariants to refine them. Work done by shankarnarayan [7], gonnord [8] deals with computation of invariants and that's why allows overapproximations of transitive closure. Fast [9] which deals with acceleration of loops, uses transitive closure. We find reachable set of states by applying transitive closure to source vertex in program verification and find whether error region is reachable or not. In equivalence checking [10] we apply transitive closure to states under consideration and check if both of them are reaching same state or not. Similarly they have prime role in maximal static expansion [11] to check weather two memory accesses are same or not. In this thesis use of transitive closure in particular for compilers those are dealing with automatic parallelization are studied.

1.3 Related work

Kelly [12] have shown that computation of transitive closure of affine tuples may not be affine in nature. Hence often we need to go for approximations. Overapproximations are considered by Beletska [13] and verdoolaege[5]. We can calculate exact transitive closure of set of relations which gives a convex set. Also exact transitive closure is computable for relations which are normalized, but that would be just subset of programs. Computation of exact transitive closure is studied by Bielecki [14] but that is for non-affine relations. Vivien maissoneuve [15] have studied comparative study of libraries used for transitive closure computation. Vivien's result shows that among Aspic [16], ISL[17] and PIPS[18] no one is better than other, they perform better on particular type of problem.

1.4 Our contribution

In this thesis, I have done comparative study of automatic compilers, which extract parallelization using different strategies. All of them generate code with OpenMP pragmas. In particular, I have considered Traco [3] compiler which uses transitive closure computation to find independent statements, and Polly [2] which uses linear/integer-linear programming based techniques to find parallel loops. Pluto [1] is also automatic parallelization tool on polyhedral model is also studied. Due to Pluto particularly works on C language and needs input program with pragma scops to show probable scope for parallelization, it is not discussed much in this thesis.

I have tested the performance of both of them using the Polybench [19] benchmarks on various metrics: For example, the results are plotted for serial and parallel execution of programs. Also improvement in Polly's parallelization extraction is suggested. Scalability issues of Traco compiler are discussed. In this thesis, i have done comparative study of automatization compilers, who tries to extract parallelization by different strategies and adds OpenMP pragmas to expose parallelization. I have particularly considered Traco compiler which uses transitive closure computation to find independent statements, and Polly which rely on iteration vector to find independent loop iterators. Performance of both of them is tested on Polybench benchmark and results are plotted for serial and parallel execution of programs. Also improvement in Polly's parallelization extraction is suggested. Scalability issues of Traco compiler are discussed.

1.5 Organisation of thesis

Remainder of thesis is organised as follows. In Chapter 2, we will see introduction to polyhedral compilation, different terms and terminology used to define it, some mathematical definitions needed to understand underlying theory. We will also have brief overview of LLVM and Polly. In Chapter 3, we will study few motivating applications which use transitive closure. We will also briefly see current work done in computation of transitive closure and their strategies used. In Chapter 4, we will have close look of Traco compiler, its functionality. In Chapter 5, we will see comparative study of different approaches used to extract parallelization and their effectiveness. We will also see scalability issues associated with them. In Chapter 6, we will have concl its functionality.

Chapter 2

Background and Definitions

In this chapter, we will see introduction to polyhedral theory in section 2.1, we will see it's usefulness in program transformation. In section 2.2 we will see basic structure considered by polyhedral compilation. In section 2.3 we define mathematical background behind polyhedral compilation.

2.1 Polyhedral Compilation

Direct translation of high level program to assembly code to object code most likely produce very inefficient code. Architectures are now a days quite complex including several levels of cache memory, many cores, deep pipelining, number of functional units, registers etc. Task of getting the best possible performance object code must utilize target architecture in most efficient way is left to compiler. As long as output program gives the same result as that of input program compiler is free to transform intermediate code in any manner. From the very first compiler intermediate representation of programs is in terms of Abstract syntax trees. ASTs represent each statement of program exactly once even if statement is going to execute many number of times due to being in a loop. This sort of representation naturally puts lots of restrictions to optimize the program.

Polyhedral compilation uses compilation technique that rely on mathematical representation of programs, especially those involving nested loops and arrays. It uses geometric and combinatorial optimization on program to analyze and optimize them. Initially it was introduced for compiler parallelizer and then adopted by wide range of applications like data locality optimization, memory management optimization, program verification etc.

Following are the most useful functionality provided by polyhedral compilation.

- It optimize or analyze program based on shape of program and not on size.
- It gives symbolic counterpart for program.
- It works on granularity of array element giving complete control over each element.

Following is the example of domain transformation of program.

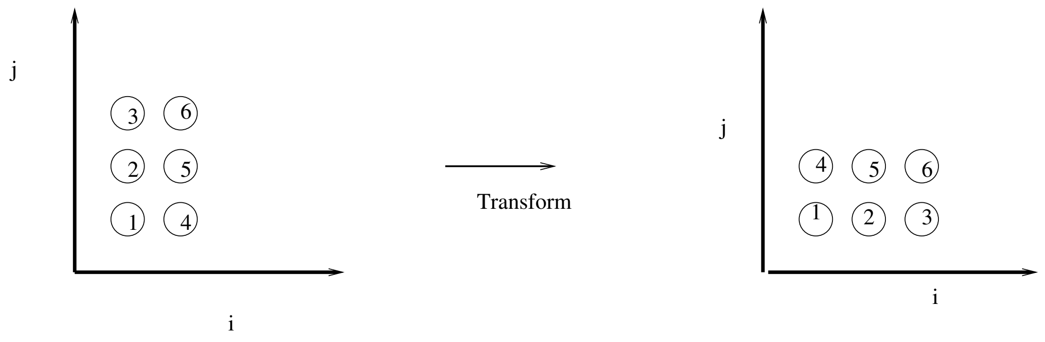

Figure 2.1: Transformation

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} -1 \\ 2 \\ -1 \\ 3 \end{bmatrix} \geq 0 \qquad \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \qquad \begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{bmatrix} -1 \\ 2 \\ -1 \\ 3 \end{bmatrix} \geq 0$$

2.2 Affine Control Loop(ACL)

Affine Control Loops are defined in [20].In affine control loop programs, there are two different classes of variables: *Data variables* and *Index variables.* The *Index variables* also include *size parameter* which is assigned only one value at any instance of program. Data variables are typically treated as multi-dimensional arrays and represents primary values computed by program. Scalars are treated as zero-dimensional arrays. Index variables are of integer type and never explicitly assigned. They get their value implicitly in execution of loop and are used to access the data variables.

The only control construct allowed is either *for loop* or *while loop* . Note that there is no if-then-else part. The body of loop is either a assignment statement, another loop or sequential combination of both. In any assignment statement left hand side(lhs) is data variable and right hand side(rhs) is expression involving data variables. Access function of data variables is affine function of surrounding loop indices.

In ACL assignment statement S executed many times depending on different values of surrounding loop indices. Loop indices are called as valid if they are within appropriate bounds. The set of valid indices surrounding S is called as *iteration domain, D.* Since there are no conditionals, each statement in loop must be executed for each valid value of index of surrounding loop. Every operation in loop is then identified by $< S_i, z >$, where S_i is statement and $z \in D$ is an integer vector, the *iteration vector.* Following is example of ACL.

```
for(i=0;i<=N;i++)
{
    for(j=0;j<=N;j++)
       {
             A[i][j] = A[i][j] + u[i] * v[j];
       }
}
```

2.3 Mathematical Definitions

In this section we will see mathematical background behind polyhedral compilation. Different types of polyhedral structure are defined in following section.

2.3.1 Rational Polyhedron vs Polyhedron

A Rational polyhedron is a subset of $\mathbb{R}^{\mathrm{n}}$defined by a finite set of inequalities. Let $P = \{x \in \mathbb{R}^{\mathrm{n}} | Ax \leq b\}$ be a polyhedron, where A and b are rational. We call rational polyhedron as a *polyhedron* when we point to set of integral points in it.

Eg.

$$P_1 = \{i, j \mid 0 \leq i \leq 4, 0 \leq 3j \leq 17\}$$

can be interpreted as a rational polyhedron or *polyhedron,* in later case it contains 30 integer points.

2.3.2 Lattice

A lattice is a subset of $\mathbb{R}^{\mathrm{n}}$defined by integral linear combination of linearly independent vectors of $\mathbb{R}^{\mathrm{n}}$, called generating vectors, plus affine vector. An integer lattice is a lattice having generating vectors and affine part as integral.

Let $B = \{b_1, b_2, ..., b_k\} \in \mathbb{R}^{\mathrm{n*k}}$ be linearly independent vectors in $\mathbb{R}^{\mathrm{n}}$. Then the lattice generated by B is given by following.

$$L(B) = \{Bx | x \in \mathbb{Z}^{\mathrm{k}}\} = \{\sum_{i=1}^{k} x_i * b_i | x_i \in \mathbb{Z}^{\mathrm{k}}\}$$

Lattice of $L_1 = \{2i + 1, 3j + 5 | i, j \in \mathbb{Z}\}$ is shown in figure 2.2 .

2.3.3 Z-Polyhedra

A Z-Polyhedra is a intersection of integer polyhedra with lattice. According to [21] alternatively Z-polyhedra can be defined as invertible affine image of integer polyhedra.

Z-Polyhedra for above lattice and integer polyhedra is given by

$$Z_1 = P_1 \cap L_1$$

$$Z_1 = \{2i + 1, 3j + 5 | -1 \leq 2i \leq 4, -15 \leq 3j \leq 2\}$$

Alternatively Z_1can be defined as image of polyhedron $Q_1 = \{i, j | -1 \leq 2i \leq 4, -15 \leq 3j \leq 2\}$ by an affine function $\{(i, j) \rightarrow (2i + 1, 3j + 5)\}$. $P_1$is obtained by taking pre-image of $Q_1$by function defining lattice i.e. $\{(i, j) \rightarrow (2i + 1, 3j + 5)\}$.

Following figures illustrate above mathematical definitions.

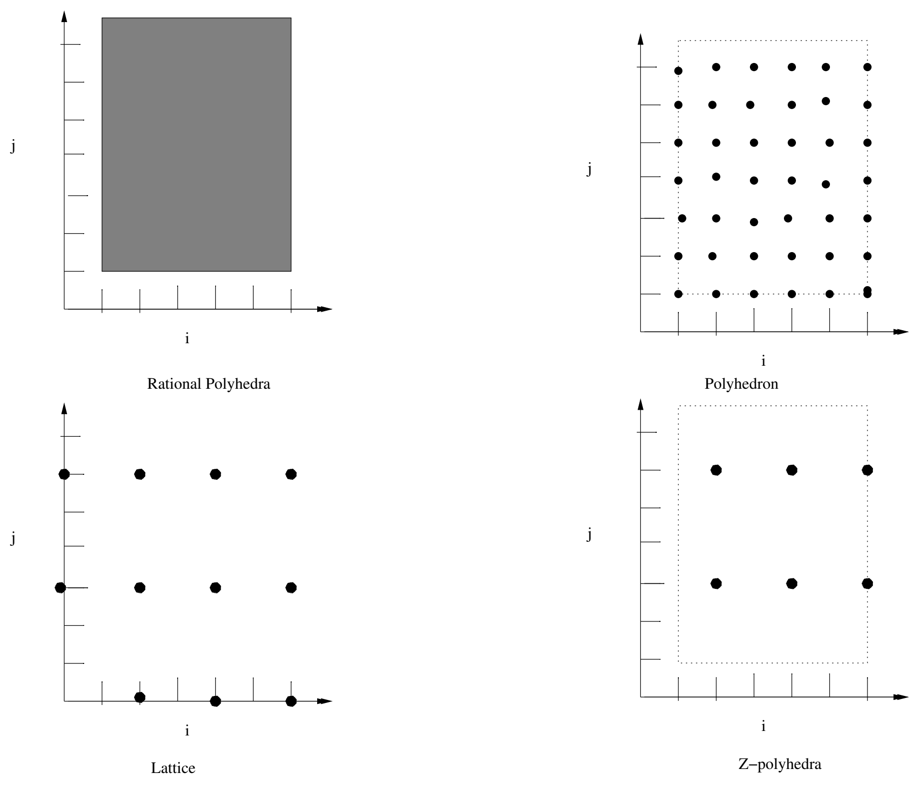

Figure 2.2: Z-Polyhedra

Facts of Z-Polyhedra

- Iteration domains are in fact a Z-polyhedra with unit lattice.
- In general intersection of Z-polyhedra is not convex.
- Union is complex to compute.
- Parametric lattices are challenging.
- We can count number of points inside, optimize and scan.

2.3.4 Affine Transformation

Affine transformation is a function which preserves points, lines and planes eg. scaling, reflection, rotation translation etc. Affine transformation are where we can multiply predicate by constant and can add a constant.

Example: $R_2 := \{[i] \rightarrow [2i] \mid 1 \leq i \leq n\}$ is an affine relation. But relation $R_2 := \left\{[i] \rightarrow \left[i^2\right] \ |1 \leq i \leq n\right\}$ is not an affine relation.

Affine relations are basically needed to reduce problem to integer linear programming.

2.3.5 Quasi-affine Integer Tuple Sets and Relations

Quasi-affine integer sets and relations are relations of the form:

$$S\left(s\right) \coloneqq \left\{x \in Z^{d} | \exists z \in Z^{e} \colon Ax + Bs + Dz \geq c\right\}$$

$$R\left(s\right) \coloneqq \left\{x1 \to x2 \in Z^{d1} \times Z^{d2} | \exists z \in Z^{e} \colon A_1 x_1 + A_2 x_2 + Bs + Dz \geq c\right\}$$

A quasi-affine relation may involve parameters which correspond to symbolic constant. In the above definition, s is a parameter. Quasi-affine term is used to specify existentially quantified variable z. Any Presburger formula can be represented in this form.The inequality of above set gives a convex set of points in d-dimensional field.

2.3.6 Powers of Relation

Power of is defined as below.

Definition (Power of a Relation) Let $R \in Z^n \to 2^{z^d \to z^d}$ be a relation and $k \in Z_{\geq 1}$ a positive number, then power k of relation R is defined as

$$R^k \coloneqq \begin{cases} R & if \quad k = 1 \\ R \circ R^{k-1} & k \geq 2 \end{cases}$$

Example For the relation,
$R \coloneqq \{x \to x + 1\}$ the k-th power is $R^k \coloneqq \{x \to x + k | k \geq 1\}$.

2.3.7 Transitive Closures

Let $R \in Z^n \to 2^{z^d \to z^d}$ be a relation and $k \in Z_{\geq 1}$ then the transitive closure R^+ of R is the union of all positive powers of R,

$$R^+ \coloneqq \bigcup_{k \geq 1} R^k$$

Example transitive closure for relation,
$R \coloneqq \{x \to x + 1\}$ can be written as $R^+ \coloneqq \{x \to y | \exists k \geq 1 \colon y = x + k\} \coloneqq \{x \to y | y \geq x + 1\}$.

2.3.8 Approximation

Fact: Even if relation is expressed in affine integer tuple form still its transitive closure and power(with parameter k) may not be affine in nature.

Example: consider a following simple relation and its power k relation.

$$R \coloneqq \{x \to 2x\}$$

$$R^k \coloneqq \left\{x \to 2^k x\right\}$$

In this example power k of relation is not affine and so is a case with transitive closure. Hence we need approximation. Two varieties of approximations are possible.

Over-Approximations (OA) OA can be used in cases like program verification. Here we can show that error state is unreachable even when reachable set is overestimated. In automatic parallelization where we can say two statements are independent even when dependence relation is overestimated.

Under-Approximations (UA) UA is particularly useful in computing communication free processes where it is often case that we obtain only one connected component.

Chapter 3

Literature Survey

In this chapter, in section 3.1, we will study few motivating applications which heavily rely on computation of transitive closure to achieve their functionality at maximum. We will also see current work done in computation of transitive closure and its approximations in section 3.2.

3.1 Motivating Applications

Three representative applications, which uses transitive closure computation are discussed in this section.

3.1.1 Iteration Space Slicing

The purpose of iteration space slicing is to partition iteration domain into program slices that are not interconnected through dependences. To reduce this problem to transitive closure we consider composition of dependence relation. Any pair that is connected through one or more applications of composition is in belong to same slice. Transitive closure thus connects each iteration to each other iteration in same slice. In Figure 1.1. Example of Iteration space slicing is explained in details in chapter 1.

3.1.2 Equivalence Checking

Both programs can be represented as inverted dependence graph that has been annotated with statements that performed in that node. The two programs are equivalent if every pair of paths that start from same element of same output arrays are such that they pass through nodes that compute same constant or in nodes that read the same elements from the same input array. This is essentially a reachability analysis, Barthou et al [10].

Derive accessibility relation from regular expression-

- concatenation composition
- branches union
- cycles transitive closure

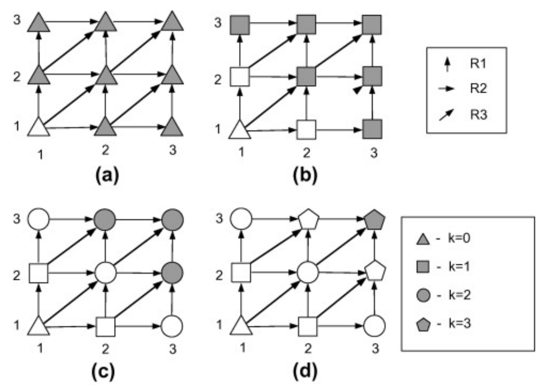

Figure 3.1: Free scheduling

image source: [3]

3.1.3 Free Scheduling

Bielecki [3] proposed an approach permitting us to build free scheduling for statement instances of affine loops. A legal schedule of loop statement instances is function that assigns a time of execution to each loop statement instance preserving all depends in loop. Under free schedules, statement instances are executed as soon as their operands are available. This allows us to minimize number of synchronization events. Basic operation of this is to compute R^k.

3.2 Current Work

In this section, we will see different approaches used to compute transitive closure.

3.2.1 Kelly et al.

Kelly et al.[12] have shown that computation of transitive closure of affine relation may not be affine or even computable. Author have modified floyd warshall algorithm to compute transitive closure. Kelly particularly focuses on computation of underapproximation, which are targeted for particular applications.

A relation is said to be in d-form iff it can be written as

$$\{[i_1, i_2, \ldots, i_m] \to [j_1, j_2, \ldots j_m] \,|\forall p, 1 \leq p \leq m, L_p \leq j_p - i_p \leq U_p \wedge j_p - i_p = M_p\alpha_p\}$$

Where L_p,$U_p$are constants and $M_p$is an integer. Transitive closure of d-form is

$$\{[i_1, i_2, \ldots, i_m] \to [j_1, j_2, \ldots j_m] \,|\exists k > 0 \; s.t. \; \forall p, 1 \leq p \leq m, L_p k \leq j_p - i_p \leq U_p k \wedge j_p - i_p = M_p\alpha_p\}$$

If relation is not in d-form then we have to go for approximation. Here we set lower bound and iteratively refine it to get better approximation.

$$R^{+}_{LB(n)} = \bigcup_{k=1}^{n} R^{k}$$

Drawback of this approximation gives large number of relations, that cannot be handled further.

3.2.2 Bielecki et al.

Bielecki et al.[14] exclusively tries to compute exact transitive closure for subset of relations that are normalized, i.e. linear in nature. Author consider relations that are not affine, we prefer affine relations cause they are easier to manipulate.

Bielecki et al. here author gives iterative algorithm which tries to compute exact transitive closure, if it not computable then it goes for overapproximations. Proposed solution is set of four algorithms. Input to algorithm is set of relations.

1. Algorithm 1: Firstly it recognizes class of each relation like d-form relation, uniform relation, relations describing chaining only, relations with different number of input and output indices etc. Calculate transitive closure for each relation separately and then take a union.
2. Algorithm 2: If exact transitive closure computation is not possible then convert relation to d-form by overapproximating it and compute transitive closure.
3. Algorithm 3: Calculate union of these transitive relations.

3.2.3 Verdoolaege, Albert Cohen et al.

Verdoolaege [5] computes difference set $\triangle$. The elements of $\triangle$ are difference in translation. $k\triangle$is path of k length in $\triangle$.

$$R := \{x \to y | y \geq 3 + x \wedge y \leq 4 + x\}$$

$$\triangle^{'} := \{d | \exists k, 3k \leq d \leq 4k\}$$

$$\left\{x \to y \in (domR \times rangR) \,|\exists d \in \triangle^{'} : y = d + x\right\}$$

Parameters:

1. Parameters can be handled as constant and then project out from $\triangle^{'}$.
2. Classify constraints
   (a) Involving only variables
   $A_1 x + c_1 \geq 0 \quad A_1 x + k c_1 \geq 0$
   (b) Involving only parameters

$B_2 s + c_2 \geq 0 \quad B_2 s + c_2 \geq 0$

   (a) Involving both variables and parameters

$A_3x + B_3s + c_3 \geq 0$

copy only those who satisfy

$\nabla \bigcap \{y - x | B_{3,j}s + c_{3,j} > 0\} = \oslash$

Basic notion here is to compute set of all differences. Then find k-length path from this set and finally project out k to get transitive closure. This paper is giving better results than others but fails when differences are affine but distances are not.

Chapter 4

Compilers for Coarse Grained Parallelization

In this chapter, we will have close look at Traco compiler which tries to extract independent iteration slices. In section 4.1 we will see internal structure and libraries which traco uses. In chapter, 4.2 we will discuss approach used by Traco to find independent iteration slices. Section 4.3 discusses the LLVM and Polly, important modification to Polly's parallelism extraction algorithm is suggested.

4.1 Traco compiler

Automatic coarse grained parallelization of loop program is of great importance in parallel computing systems. Traco compiler tries to extract available parallelism in arbitrary nested program loops.

Traco is a source to source transformer compiler. It takes valid C program as a input and find its independent slices that can be executed in parallel. The resultant program will have OpenMP pragmas explaining parallelism. Traco automatically searches for loops in input program and replace them with appropriate ones. It uses Petit[22] tool for dependence analysis and Omega[23] for presburger arithmetic calculation. Traco can use ISL[24] or CLOOG[25] instead of codegen functionality of omega. Basic function blocks of compiler are shown in following fig.

4.2 Optimization

The input to Traco compiler is any valid C program. For Traco compiler, pragmas are not needed to show existence of loops. Basic functionality of finding loops is performed by function find_loops.py. It gives line number where Petit starts to extract dependence relations. Output of dependence analysis by Petit is stored in intermediate file in a form OpenScop Specifications. To find the source of dependences(Ultimate Dependence Source) from where independent slices starts Traco uses following formula.

$$UDS = Range(R) - Domain(R)$$

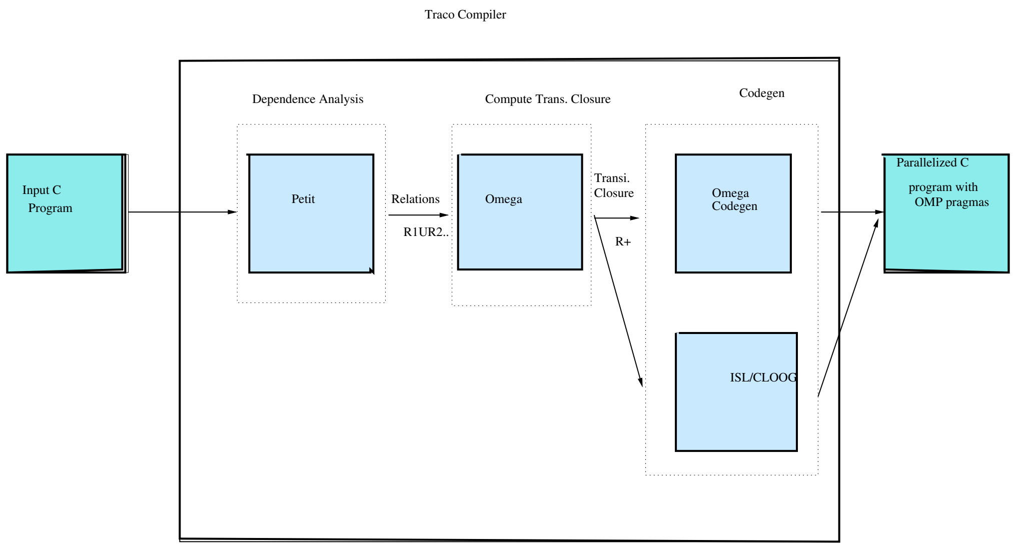

Figure 4.1: Traco Blcok Dia.

Preprocessing is done on input relations to make their input and output tuple have exactly same number of elements in each relation.

eg. Replace the tuple $e = [e_1, e_2, ...e_{m-k}]$ by $e = [e_1, e_2, ...e_{m-k}, -1, -1, -1(k\ times)]$. Also extend the input and output tuples by adding identifiers to tuple eg. $R_{i,j} = [(e) \rightarrow (e^{'})]$ by $R_{i,j} = [(e, i) \rightarrow (e^{'}, j)]$.

Dependence relations along with UDS are passed to Omega calculator. Omega applies transitive closure on each UDS and finds iteration slices. According to [26] if we have R^k then we can obtain R^+ from it and vice a versa. When we compute set $S(k) = R^k(UDS)$ at that time set can contain vertices that are reachable after k composition of relation or k length path, but a particular vertex may have more than one incoming edge having reachable path of more than k length. So to compute set of vertices that can be executed at time k we have

$$S(k) = R^k(UDS) - R^+ \circ R^k(UDS)$$

4.3 Integration of Traco's functionality in Polly

In this section,

4.3.1 Low Level Virtual Machine(LLVM)

LLVM is a compiler infrastructure developed by Lattner and Vikram adve [27]. It is build around LLVM-IR and comes with large set of optimization and transformation passes. It uses SSA based strategy and provide a middle layer of compiler system. LLVM is especially designed to optimize compile time, link time, and run time, so we need not to postpone the optimization till end. Major feature of LLVM is it's low level intermediate representation(LLVM-IR) which captures very minute details like size and type of variable. Such details would be much useful to perform effective optimizations. Tools are already available to convert high level languages like C, python to LLVM-IR.

It gives 3 times better performance than GCC.

4.3.2 Polly

Polly [28] project uses the polyhedral compilation strategy to optimize the LLVM-IR. It operates on LLVM-IR there by increasing number of beneficent languages. It is high level data locality and loop optimisation infrastructure for LLVM. It uses abstract mathematical representation of integer polyhedra to analyse and optimize the memory accesses. Polly project of LLVM currently implements auto-parallelism and vectorization. It is built around advanced polyhedral library with full support for existentially quantified variables and include its own dependency analysis. Due to simple file interface it is possible to apply transformation manually or as an external optimizer. Unlike other Polyhedral transformation Polly do not change domain, it operates only on schedule.

4.3.3 Current working and Improvements proposed

Polly is designed as compiler's internal analysis and optimization passes. Transformation in Polly would create parallel loops with OpenMP pragmas as if user have added pragmas manually. The dependency analysis module of Polly automatically detects existence of SCoPs and give them to OpenMP code generation module.

Naive Approach used to detect parallelism is to check if certain dimension of iteration space is carrying any dependence. If it is not then the dimension is parallel. This is very naive approach and can detect only fully parallel dimensions. But while generating AST, compiler may split loop over several *for* loops. This may happen automatically when cloog tries to optimize flow control. Approach used in Polly is after generating AST, analysing for each *for* loop is that can be parallelized. This is achieved by limiting normal parallelization check to subset of iteration space enumerated by the loop. Polly obtain this subset directly from cloog hence it don't need to traverse AST. This procedure of detection of parallelization is too naive can be replaced by above.

Comparison of effectiveness of Traco and Polly is tested on polybench benchmarks. Results of comparison are shown in chapter 5. Clearly Traco performs better than Polly. Both Traco and Polly adds OpenMP pragmas to expose parallelism. Basic difference between their functionality is there ability to extract dependences. Traco internally uses omega to compute transitive closure of dependence relations. Even Polly has ISL inbuilt it do not use it. Omega was developed around 90's, now a days bit outdated and is not maintained any more, also there are many corner cases which are not handled properly in it. While ISL is inspired by omega and well tested and maintained. So there are large benefits to implement above advanced algorithm in Polly and compute transitive closure using ISL calls.

Chapter 5

Experiments with Two Polyhedral Compilers

In this chapter, we will see comparative study of Traco and Polly and their scalability. Section 5.1 shows experimental input and output by Traco compiler. Section 5.1.1 shows speed up achieved by Traco over serial computation. Section 5.1.2 shows scalability issues of Traco. Section 5.2 gives Polly's speed up achieved.

5.1 Traco

Polybench is a benchmark containing static control parts. It has feature like non-null data initialization, syntactic data constructs to prevent dead code elimination, parametric loop bounds for general purpose implementation and clear kernel marking using #pragma scops. It has programs like two matrix multiplication, LU decomposition, dynamic programming, seidel, cholesky decomposition, 2D image processing etc.

Following is a piece of code form dynprog.c of Polybench benchmark. It is transformed by Traco to extract parallelism.

```
#pragma scop
 for (t = 0; t < niter; t++)
 {
        for (j = 0; j <= maxgrid - 1; j++)
                for (i = j; i <= maxgrid - 1; i++)
                for (cnt = 0; cnt <= length - 1; cnt++)
                        diff[j][i][cnt] = sum_tang[j][i];
     for (j = 0; j <= maxgrid - 1; j++)
     {
                for (i = j; i <= maxgrid - 1; i++)
         {
                        sum_diff[j][i][0] = diff[j][i][0];
               for (cnt = 1; cnt <= length - 1; cnt++)
                                sum_diff[j][i][cnt] = sum_diff[j][i][cnt - 1] + diff[j][i
                     mean[j][i] = sum_diff[j][i][length - 1];
         }
     }
     for (i = 0; i <= maxgrid - 1; i++)
                path[0][i] = mean[0][i];
```

```
    for (j = 1; j <= maxgrid - 1; j++)
        for (i = j; i <= maxgrid - 1; i++)
              path[j][i] = path[j - 1][i - 1] + mean[j][i];
  } #pragma endscop
```

Result of transformation is shown below.

```
#pragma scop
for (t = 0; t < niter; t++)
{
        #pragma omp parallel for
    for (j = 0; j <= maxgrid - 1; j++)
                #pragma omp parallel for
                for (i = j; i <= maxgrid - 1; i++)
                        #pragma omp parallel for
                        for (cnt = 0; cnt <= length - 1; cnt++)
                                diff[j][i][cnt] = sum_tang[j][i];
    for (j = 0; j <= maxgrid - 1; j++)
    {
                for (i = j; i <= maxgrid - 1; i++)
            {
                sum_diff[j][i][0] = diff[j][i][0];
                        #pragma omp parallel for
               for (cnt = 1; cnt <= length - 1; cnt++)
                        sum_diff[j][i][cnt] = sum_diff[j][i][cnt - 1] + diff[j][i][cnt];
                      mean[j][i] = sum_diff[j][i][length - 1];
        }
     }
        #pragma omp parallel for
    for (i = 0; i <= maxgrid - 1; i++)
                path[0][i] = mean[0][i];
    for (j = 1; j <= maxgrid - 1; j++)
                for (i = j; i <= maxgrid - 1; i++)
                          path[j][i] = path[j - 1][i - 1] + mean[j][i];
 }
#pragma endscop
```

5.1.1 Runtime Experiments

Benchmarks from polybench are passed through Traco to extract the available parallelism. Execution time for both original benchmark and transformed benchmark are plotted in Figures 5.1,5.2 and 5.3.

Execution time are in seconds shown on y-axis. To note execution time -O3 option is used in for both original as well as transformed code. Results are taken on 32 gpu nvidia machine.

Speed up achieved is depends two factors, primarily it depends on available parallelism in program and secondly on compilers ability to extract it. Hence different benchmark shown different level of speedups. For adi.c which is Alternating Direction Implicit solver, Traco was unable to parallelize code because Petit was not able to extract dependence from it. For some benchmark transformed code is taken more time than original one. This is due to -O3 optimization used for compiling. These examples shows few instances where -O3 optimization produce code which is even more time consuming. Polybench benchmarks differ in runtime in a large margin. So benchmarks having nearly same runtime are grouped in one figure. Total three plots are shown for thirty benchmarks.

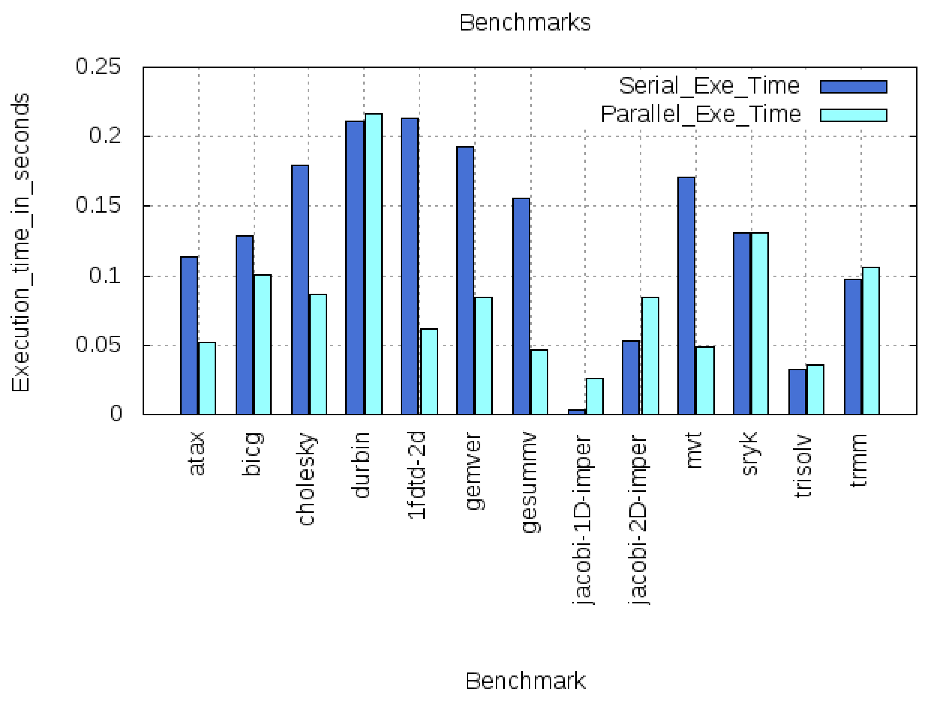

Figure 5.1: Traco 1

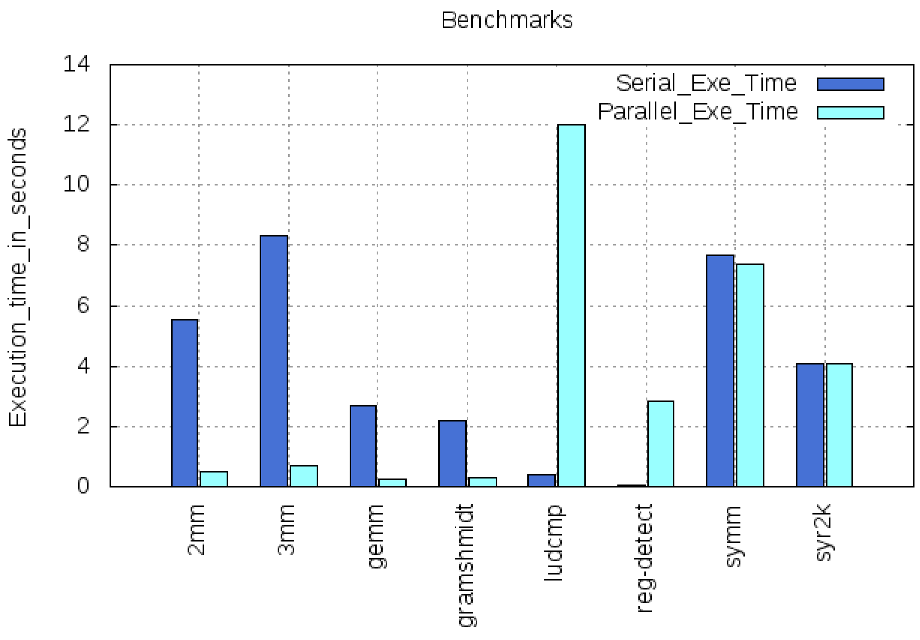

Figure 5.2: Traco 2

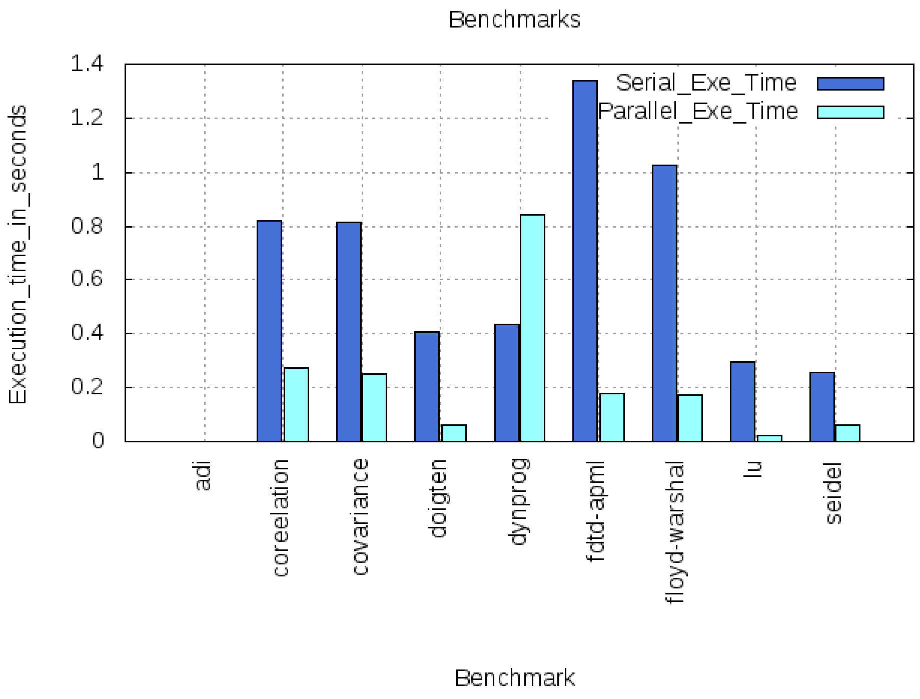

Figure 5.3: Traco 3

5.1.2 Compile Time Experiments

Traco's performance is largely dependent on dependence analysis. If there are large number of dependences that cannot be handled by omega, then its performance degrades. Scalability of Traco is tested on matrix multiplication programs. Number of matrices are multiplied in a careful manner so that to guard against trivial optimizations and any dead code elimination. The are multiplied in manner shown below. Traco is unable to handle dependence relations after 20 matrix multiplication and fails to transform the code. In figure 5.4 x-axis shows number of matrices multiplied and y-axis shows execution time of original code and parallelized code.

```
          D=AB      //D,A,B are matrices
          F=DE
          H=FG
          ...
```

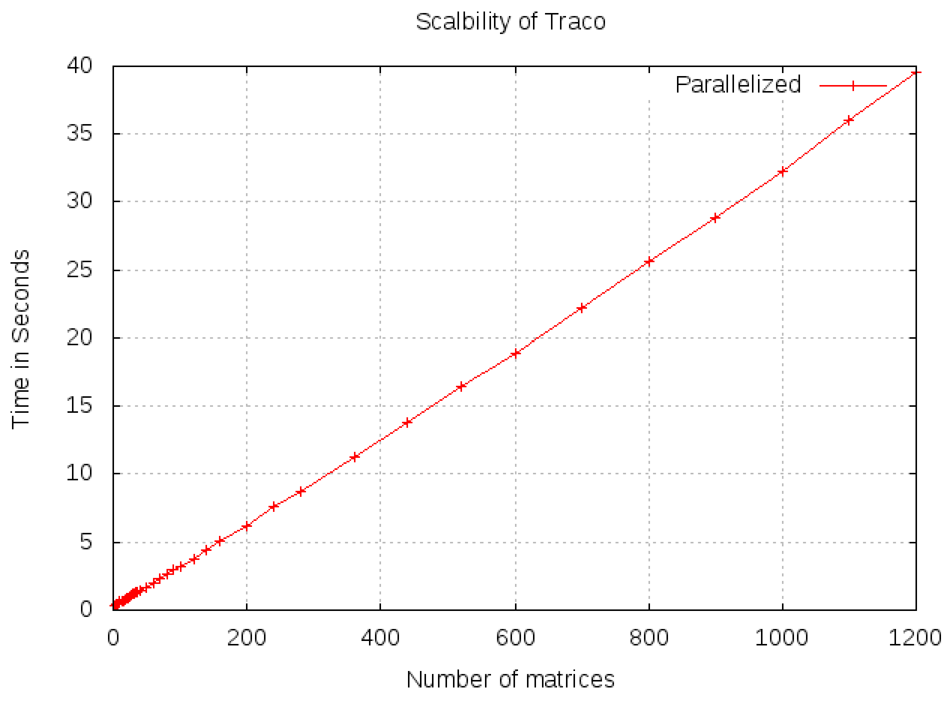

Figure 5.4: Un-scalability of Traco

As Petit was able to extract dependences for 2 matrix multiplication hence unscalability is not due to inability to extract dependences. The reason for unscalability is Omega's inability of handle large number of dependence relations to compute transitive closure. The main point to note is scalability of computation of transitive closure affects the overall performance of compiler in a large manner.

5.2 Polly

Polly's performance is tested on Polybench benchmark version 3.2. Improvement in execution time of benchmarks is shown in figure 5.5, 5.6 and 5.7. Performance of Polly is quite less than that of Traco, this is directly due to inability to extract maximum parallelism from available parallelism. Polybench benchmarks differ in runtime in a large margin. So benchmarks having nearly same runtime are grouped in one figure. Total three plots are shown for thirty benchmarks. X-axis shows benchmarks under consideration and y-axis shows execution time in seconds. Experiments were taken on 1.2MHz 64 bit 4 cpu system.

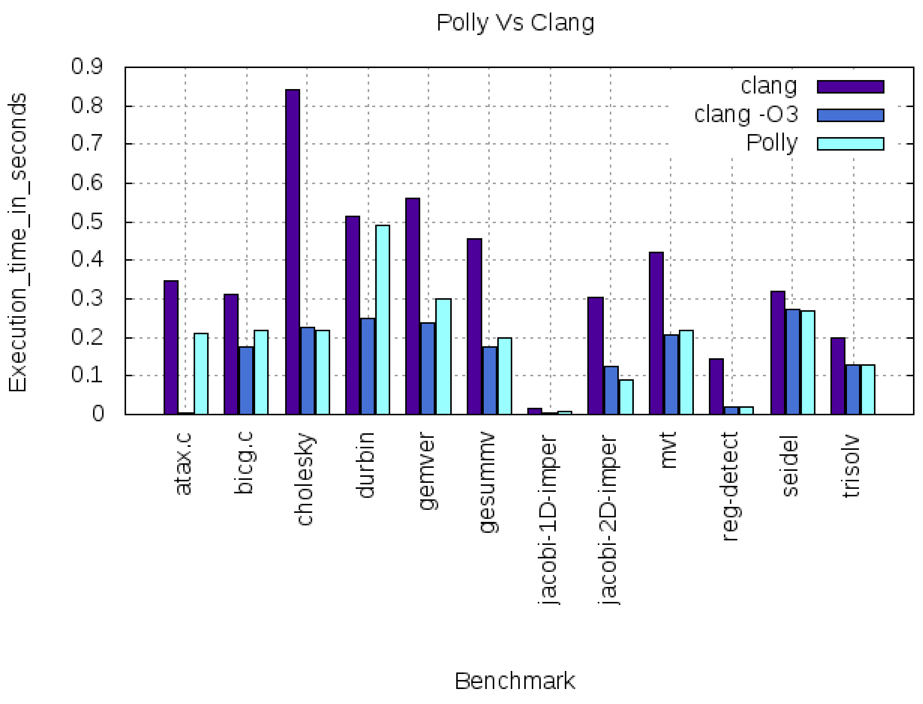

Figure 5.5: Polly 1

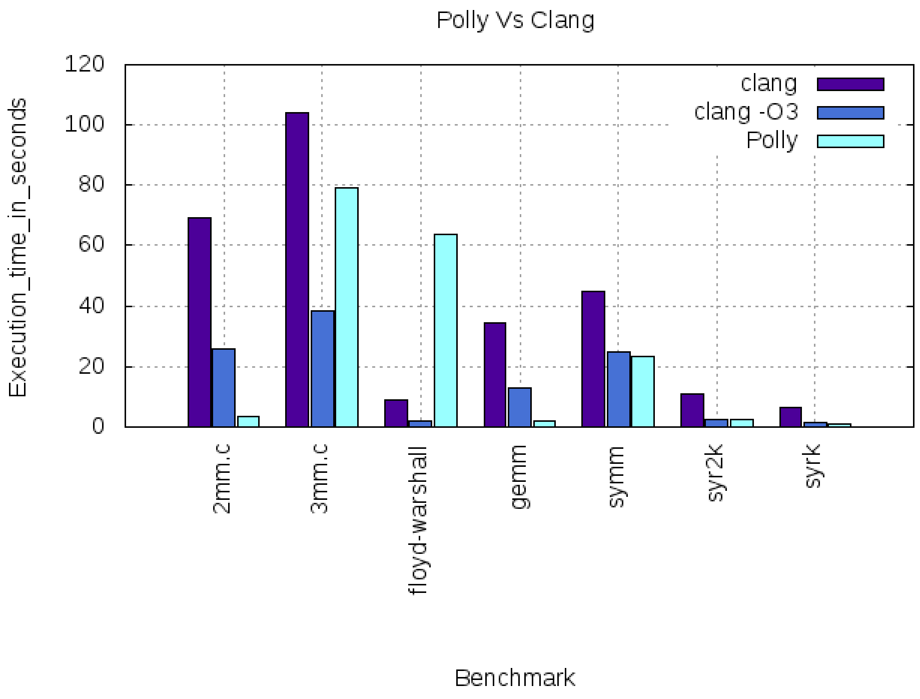

Figure 5.6: Polly 2

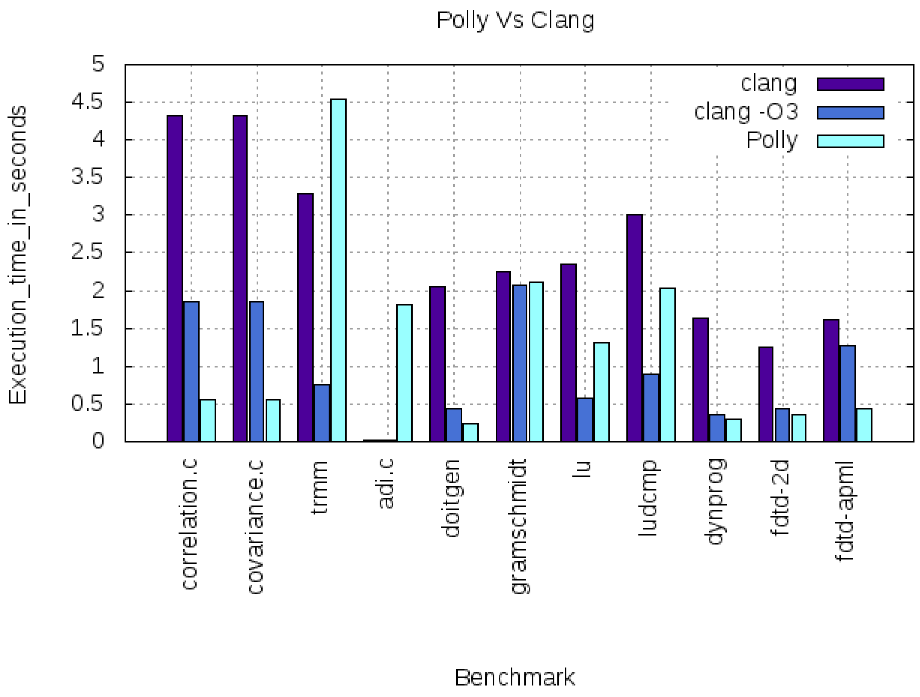

Figure 5.7: Polly 3

Chapter 6

Conclusions and Future work

In this chapter, we will conclude by giving conclusion and showing future direction in which this work can be extended.

Conclusions In this thesis, we particularly focuse on ability of compiler to extract parallelism from available parallelism in program. We studied different approaches used in compilers like Traco and Polly. We saw the importance of transitive closure computation on compiler's ability to extract coarse grained parallelism. We show that transitive closure based approaches extract parallelism in better way when compared to linear programming techniques. We also show the drawbacks of Transitive closure based approaches based on scalability issues. We suggest new algorithm for Polly so that it can overcome it's drawbacks.

Future Work

- There is huge scope of implementing transitive closure based parallelism extraction algorithm in LLVM. LLVM have a large set of inbuilt functionality that can add up performance and help in implementation too.

- The algorithm for computation of Transitive closure could be improved, which should give better approximations, both overapproximations and underapproximations, and should also scale well.

- We can use different representation for showing dependence and transitive closure, like Z-Polyhedra, where computation of transitive closure could be more accurate and scalable.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43, (2008) 101–113.

[2] T. Grosser, A. Größlinger, and C. Lengauer. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22. `http://polly.llvm.org/`.

[3] W. Bielecki, M. Palkowski, and T. Klimek. Free scheduling for statement instances of parameterized arbitrarily nested affine loops. *Parallel Computing* 38, (2012) 518 – 532.

[4] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, (1998) 46–55.

[5] S. Verdoolaege, A. Cohen, and A. Beletska. Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. In Proceedings of the 18th International Conference on Static Analysis, SAS'11. Springer-Verlag, Berlin, Heidelberg, 2011 216–232.

[6] S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Towards a Model-checker for Counter Systems. In Proceedings of the 4th International Conference on Automated Technology for Verification and Analysis, ATVA'06. Springer-Verlag, Berlin, Heidelberg, 2006 493–507.

[7] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In Static Analysis, 53–68. Springer Berlin Heidelberg, 2004.

[8] P. Feautrier and L. Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. In Tools for Automatic Program AnalysiS. Perpignan, France, 2010 .

[9] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer* 10, (2008) 401–424.

[10] D. Barthou, P. Feautrier, and X. Redon. On the Equivalence of Two Systems of Affine Recurrence Equations (Research Note). In Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Euro-Par '02. Springer-Verlag, London, UK, UK, 2002 309–313.

[11] D. Barthou, A. Cohen, and J.-F. Collard. Maximal Static Expansion. *Int. J. Parallel Program.* 28, (2000) 213–243.

[12] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive Closure of Infinite Graphs and Its Applications. *Int. J. Parallel Program.* 24, (1996) 579–598.

[13] A. Beletska, D. Barthou, W. Bielecki, and A. Cohen. Computing the Transitive Closure of a Union of Affine Integer Tuple Relations. In D.-Z. Du, X. Hu, and P. Pardalos, eds., Combinatorial Optimization and Applications, volume 5573 of *Lecture Notes in Computer Science*, 98–109. Springer Berlin Heidelberg, 2009.

[14] W. Bielecki, T. Klimek, and K. Trifunovic. Calculating Exact Transitive Closure for a Normalized Affine Integer Tuple Relation. *Electronic Notes in Discrete Mathematics* 33, (2009) 7 – 14. International Conference on Graph Theory and its Applications.

[15] V. Maisonneuve, O. Hermant, and F. Irigoin. ALICe: A Framework to Improve Affine Loop Invariant Computation *. In the 5th International Workshop on Invariant Generation (WING 2014) . Vienne, Austria, 2014 `http://www.cri.ensmp.fr/people/maisonneuve/alice.git`.

[16] L. Gonnord. Aspic Version 3.3 2010. `http://laure.gonnord.org/pro/aspic/aspic.html`.

[17] S. Verdoolaege. ISL-Integer set library 2015. `http://repo.or.cz/w/isl.git`.

[18] P. Feautrier and L. Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2Fsm. *Electron. Notes Theor. Comput. Sci.* 267, (2010) 3–13.

[19] L. N. Pouchet. PolyBench: The Polyhedral Benchmark suite.

[20] S. V. Rajopadhye. Dependence Analysis and Parallelizing Transformations. In The Compiler Design Handbook, 329–372. 2002.

[21] G. Gupta and S. Rajopadhye. The Z-polyhedral Model. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '07. ACM, New York, NY, USA, 2007 237–248.

[22] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. New User Interface for Petit and Other Extensions. *User Guide* 1, (1996) 996.

[23] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Calculator and Library, version 1.1.0 1996. `http://www.cs.umd.edu/projects/omega/`.

[24] S. Verdoolaege. Isl: An Integer Set Library for the Polyhedral Model. In Proceedings of the Third International Congress Conference on Mathematical Software, ICMS'10. Springer-Verlag, Berlin, Heidelberg, 2010 299–302. `http://repo.or.cz/w/isl.git`.

[25] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04. IEEE Computer Society, Washington, DC, USA, 2004 7–16.

[26] A. Beletska, D. Barthou, W. Bielecki, and A. Cohen. Computing the Transitive Closure of a Union of Affine Integer Tuple Relations. In D.-Z. Du, X. Hu, and P. Pardalos, eds., Combinatorial Optimization and Applications, volume 5573 of *Lecture Notes in Computer Science*, 98–109. Springer Berlin Heidelberg, 2009.

[27] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04. IEEE Computer Society, Washington, DC, USA, 2004 75–.

[28] T. Grosser et al. Polly - LLVM Framework for High-Level Loop and Data-Locality Optimizations 2015. `http://polly.llvm.org/`.