A Framework for Generation of Parallel Transformers from Specifications

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

Master of Technology

by
Amruta Adewar
Roll Number : 12111005

under the guidance of
Dr. Amey Karkare
and
Late Dr. Sanjeev Kumar Aggarwal

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
June, 2014

CERTIFICATE

It is certified that the work contained in this thesis entitled
"*A Framework for Generation of Parallel Transformers from Specifications*", by *Amruta Adewar(Roll Number 12111005)*,
has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Amey Karkare)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

June, 2014

Abstract

Code optimization or code transformation is an important phase in the process of compiler construction. It is a complex procedure as it involves analysis of the entire program. We propose a framework which assists in writing parallel transformations and reduces efforts of manually writing these code transformations. The framework consists of the specification language and a transformer generator tool. The specification language provides methods for dependence analysis and for manipulating concerned code elements. The transformer generator tool works similar to the tools such as Lex and Yacc used for generating Lexical Analyzers and Parsers. The tool takes specifications as an input and generates transformer from it.

With the growing interest in the field of parallelization and High Performance Computing (HPC), a lot of research has been going on in experimenting and introducing new techniques which extract parallelism from the sequential programs. Experiments are also needed to be done to decide the proper ordering of different techniques to get the maximum benefit. Our framework will be useful with this experimentation. Researchers and compiler writers can write specifications for different transformations and can test their effects using the generated transformers. It is easy and time-reducing to write in the specification language as the length of the specification is quite less as compared to the actual size of the transformations.

Dedicated to

my family, my advisors and my friends.

Acknowledgement

I would like to thank my thesis supervisor, ***Dr. Amey Karkare*** *for his tremendous support and invaluable suggestions in giving a proper direction to my efforts. I feel motivated and encouraged every time I attend his meeting. I am grateful to my advisor,* ***Late Dr. Sanjeev Kumar Aggarwal*** *for providing me with the opportunity to work on this topic. His insights into my research topic helped me better understand the field of compiler optimization.*

I would also like to thank all the faculty members of Computer Science and Engineering Department for imparting with invaluable knowledge and adding a lot of value-oriented growth to my career. I would like to express my gratitude towards the entire CSE department and IIT Kanpur for providing me with all the facilities and proper environment.

I thank all my friends who made my college life all the more cheerful and memorable. Their discussions always motivated me during the thesis work. I will miss the outings, the late night discussions and all the fun that we had over this period of two years. I acknowledge Akanksha for reading the draft of the thesis and suggesting improvements.

I shall forever remain in debt of my family for their unconditional love. Their guidance has helped me grow as a person and their support has been the shining light in my life. I shall always strive to be the reason of your happiness.

Amruta Adewar

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we present our work on developing a framework for generation of parallelizing transformers. The framework helps in writing formal specifications and generation of code for compiler transformations. In the literature, compiler transformations are broadly classified into two classes- scalar and parallel transformations. *Scalar transformations* mostly aim for reducing number of instructions or size of a program. Eg. Dead Code Elimination and Constant Propagation. *Parallelizing transformations* try to improve locality (spatial or temporal) in a program. In other words, they try to change the dependence relations which restrict parallelism in a program. Eg. Loop Interchange and Loop Distribution.

However, we focus our work on parallel transformations. The transformations are formally specified with the help of dependence relations.

1.1 Motivation

Code optimization is an important phase in compiler constrution. In this phase, code improving transformations are applied on a code to improve its performance. The main aim of these transformations is to reduce the size or execution time of a program. The phase has significant impact on high performance computing systems.

Many performance improving scalar and parallelizing transformations are described in the literature. However, there are some important points which are to

be taken into consideration while applying these transformations to a program. A thorough dependence analysis of a program is required to find out whether applying a transformation is legal or not because transformation should not change the semantics of a program. The dependence analysis techniques used for this purpose are often time-consuming. Also, the order in which transformations are applied has a huge impact on the performance of the optimized code. Since, there cannot be a well defined method that helps in deciding which optimizations to apply and where, experimentation is essential to find out the best possible set or sequence of transformations for a particular program. To help out in these issues, a tool which reduces efforts of manually writing code optimizers can be much beneficial.

Lex and Yacc are some of the most widely used tools in the areas of compiler construction, and form an integral part of most of the modern compiler infrastructures. These tools have greatly reduced the efforts of manually writing Lexical analyzers and Parsers. Tools are also available to assist code generation. However, no such tool is available to help in code optimization.

Our motivation behind this thesis was to develop a tool or a framework which can help compiler writers in writing code optimizers. Such a tool can help researchers and compiler writers in studying properties, performance and interdependencies of different transformations, finding out a proper or beneficial sequence of transformations on a specific program set and for experimenting with some new transformations.

1.2 Related Work

Whitfield and Soffa [1, 2, 3], Paleri, V. [4], and Karuri, K. [5] have worked on formally specifying compiler optimizations and generating optimizers from the specifications. Dependence relations are the basis for writing specifications in all of these works.

Whifield and Soffa, designed a specification language, GoSpeL (General Optimization Specification Language) and an optimizer generator tool, GENensis. The language has clear and easy semantics as sentential forms are used for writing specifications. For each optimization, GOSpeL requires specification of type, precondition

and action to optimize the code. The type section specifies the types of required code elements. Precondition consists of code patterns and dependence information. Code patterns specify format of the code whereas dependence information consists of global information about control and data dependences required for the specified optimization. The action part consists of primitive actions which collectively give effect of the optimizing transformation to be applied. The language is more conservative as we cannot write subspecifications or nested conditions. Also, specification of all the three parts are essential which is not feasible for every optimization. They have generated many scalar and parallel transformations using the tool, GENesis.

Paleri, V. has designed a specification language which follows the formal (Q:R:P) notation where, Q stands for quantifiers, R is a valid formula and P specifies precondition. This is followed by the action part specifying primitive actions. The precondition is specified using the same (Q:R:P) notation, allowing the nesting of preconditions. Paleri's language is less conservative as it offers flexibility in writing condition and action part and allows nested conditions. However, the language is more formal and thus has a slightly high learning quotient. The language constructs are not easy to understand.

Karuri's language uses hybrid approach. The language constructs are in the setential form modelled using the formal (Q:R:P) notation. He has designed a tool, OptGen to generate optimizers.

Both Paleri, V. and Karuri, K. have focussed on scalar transformations only.

1.3 Contribution

We have developed a system in which specifications are written using the specification language. It takes specification as an input and generates the transformers for the parallelizing optimization to be performed. These transformations, when applied on a source code, transforms it as per the specification. The scope and features of the system include the following.

1.3.1 Intermediate Representation

The input program is converted into an intermediate representation. All the transformations are performed on the intermediate code. So, choosing an appropriate intermediate representation is important. We have used Abstract Syntax Tree (AST) as an intermediate representation.

1.3.2 Dependence Analysis

Preconditions are specified in the specifications in the form of dependence relations. So, the input program is analyzed for data dependence and control dependence.

1.3.3 The Specification Language

Karuri, K. has explored scalar optimizations using the specification language designed. As we have worked on parallel transformations, we modified his language and added some constructs to it required to specify parallel transformations. The details of the language are given in 4. We have specified some loop transformations using the specification language.

1.3.4 Transformer Generator

We have designed and developed a tool which generates the transformer based on the specifications read. It checks if the preconditions specified in the specification are satisfied by the input program. If it does, actions specified in the specification are performed. So, the tool is called transformer generator tool. The tool supports generation of parallelizing transformations, specifically.

The system does not parallelize the transformed code. A parallelizing technique can be used to get the desired results.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows.

Chapter 2: In this chapter, we give the Literature Survey done during the course of the thesis work.

Chapter 3: In this chapter, we highlight the major background behind this thesis. It describes some concepts which are sort of pre-requisites in writing specifications for optimizations and understanding our tool.

Chapter 4: This chapter gives the overview of the specification language we have used, along with two examples of specifications.

Chapter 5: This chapter gives the overview of the tool that generates parallel transformers.

Chapter 6: In this chapter, we discuss the experiments performed and the results obtained.

Chapter 7: In this chapter, we conclude the thesis while giving the possible extensions associated with the thesis.

Chapter 2

Literature Survey

In this chapter, we present the Literature Survey done during the course of the thesis work. Firstly, some classical books available in this area are enlisted. First section contains a list of the articles published recently about different optimizing transformations and automatic parallelization techniques. The previous work done on the approach used in the thesis is already explained in section 1.2. The last section of this chapter describes some articles presenting different other approaches of generating and experimenting with the transformations.

Different program analysis techniques are discussed thoroughly in the literature. Some code improving transformations are also described. Aho, Ullman and Sethi [6] describe traditional scalar transformations and different approaches to data flow analysis. Wolfe [7] describes data dependence analysis and some parallelizing transformations. Muchnick [8], gives brief description of both scalar and parallelizing transformations. Banerjee [9] discusses parallelizing transformations and different data dependence analysis techniques. Allen and Kennedy focus [10] on analysis and transformation techniques for parallel architectures. They cover parallelizing transformations and their interactions, in detail.

2.1 Compiler Optimization and Auto-Parallelization

Some recent work in the area of compiler optimizations and different automatic parallelization techniques are described below.

Liu et al. [11] present an iteration-level loop parallelism technique for executing different iterations of the same loop (serial or DOACROSS) in parallel. The aim is to maximize the number of consecutive iterations that are independent of each other and can be executed parallely. So, the statements are rearranged to migrate dependencies acroos the different iterations of the same loop. A dependence graph is created and dependecies are migrated among the edges of the graph using the technique of 'Retiming'. In this way, loop optimization problem becomes the graph optimization problem. Finally, a transformation algorithm which uses these retiming values is described to generate optimized code for the loop into consideration. The optimized code contains prologue, epilogue and loop which can now be parallelized.

Jie et al. [12] propose a nested loop fusion algorithm which performs cost analysis of parallel loops. It ensures that the positive speedup is gained for the loop on which Loop Fusion has been applied. The cost analysis module analyzes iteration sum of a loop. The technique fails to give positive speed-up for the loops having relatively small iteration sum trip. Loop Unrolling is applied on such loops to make them serial. This decreases the depth of the nested loop which can avail application of loop fusion.

Oancea et al. [13], introduce an Automatic Loop Parallelization technique. The technique is based on both static and dynamic data dependence analysis. Comparatively, dynamic data dependence analysis techniques are less conservative whereas, the execution overhead in static analysis is less. So, this paper proposes a hybrid compiler technology that extracts maximum possible static information to reduce cost of the dynamic analysis. The technique is better suited to parallelize larger loops, but is less effective in driving code transformations such as loop interchange, skewing, tiling, etc.

Pouchet et al. [14] propose an optimization technique based on the polyhedral model. The polyhedron is pruned, focussing on multidimensional statement interleavings corresponding to a generalized combination of loop fusion, loop distribution and code motion. In this technique, all possible dimensions and their interleavings are explored and different optimizing program schedules are selected. The algorithm then selects the best performing schedule.

hun Eom et al. [15] present a dynamic parallelization approach - Dynamic Out-of-Order Java (DOJ). DOJ is the improvement suggested over OoOJava which is a task-based dataflow programming model. DOJ handles two types of dependencies, variable dependencies and heap dependencies. The main achievement of DOJ is in compilation time. Due to the dynamic approach used in DOJ, it provides much faster compilation whereas, a comparable speedup is obtained as in OoOJava.

Noll et al. [16] propose a design of a concurrency-aware JVM for the JIT compiler. It performs runtime task size optimizations. The optimization presented here increases the parallel task size. It performs concurrent region merging of multiple small concurrent regions into a larger concurrent region. The technique turns out to be efficient for the applications which execute many smaller tasks.

2.2 Automatic generation of Transformations and experimentation

Some approaches of automatic generation of optimizing transformations and to perform experimentation with different transformations are listed below.

Farnum, Charles [17] proposes a framework, Dora to support experimentation with compiler optimizers. The framework uses a single Intermediate Langguage schema to represent all languages. It applies existing optimizations to a new combination of source language and target machine. Using Dora, it is possible to write new optimizations in the reusable manner. So, Reusability and Extensibility are the two achievements of the framework.

Tiwari et al. [18] describe their framework for auto-tuning of optimizations. The framework uses a search algorithm which evaluates different combinations of compiler optimizations and selects the one with the best performance. It supports automated code transformation and parameter search.

Thees et al. [19] present a model for automatic generation of implementations from formal specifications. Estelle is a formal description technique which is designed for the description of distributed, concurrent systems, in particular communication protocols. The model consists of an Estelle compiler which can also be used as a platform for different purposes such as performance monitoring, optimization and to test implementation methods.

Davidson et al. [20] describe a system that automatically generates patterns for a fast classical peephole optimizer. In the system, a retargetable machine-directed optimizer is run at compile-compile time. Using its output the classical compile-time peephole optimizer is automatically generated. In this way, the system achieves thoroughness and retargetability of a machine directed peephole optimizer along with the speed of a classical peephole optimizer.

Bansal et al. [21] propose the automatic generation of peephole superoptimizers. Superoptimizers find the optimal code sequence for a single, loop-free target sequence. For several different target machines, optimal instruction sequences are computed. An indexed optimization database is used to save all the optimizations found by the superoptimizer. It reduces the efforts of computing optimization again and again.

Tate et al. [22] use proof of equivalence between the original and transformed concrete programs, to generate compiler optimizations. Information such as which are the important aspects of the program and which can be discarded can be inferred from these proofs. These proofs are generated by translation validation or a proof-carrying compiler. The system accepts concrete examples of transformations as an input and from the proofs of these instances derives optimization rules.

Chapter 3

Background

This chapter will focus on some of the key concepts which are pre-requisites in writing formal specifications of transformations. We will briefly discuss different parallelizing transformations and their constraints. The chapter is concluded with a discussion on ROSE infrastructure which has been used in building our tool.

3.1 Key concepts in Compiler Optimizations

If a code is produced just by applying straightforward compiling algorithms, there is often an ample scope of improving the code in terms of its execution time or memory requirements, or both. Traditionally, program transformations which try to improve the program code are called ***Optimizations***. However, the word "optimization" is a misnomer as only in rare cases we get an object code which is optimal, by any measure. Compilers that apply code-improving transformations are called ***Optimizing Compilers***. The transformations provided by an optimizing compiler should satisfy following properties:

- A transformation must preserve the meaning of an input program. Sometimes it is desirable to take a conservative approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- A transformation must speed up programs by a measurable amount.

- A transformation must be worth the effort. In some cases, transformations can only be applied after detailed, often time-consuming, analysis of the source program. For those transformations, the improvement gained is neutralized by the resources consumed in acheiving it.

Transformations are broadly classified into two classes:

1. *Scalar Transformations* aim at reducing the size of a program.

2. *Parallelizing Transformations* aim at maximizing parallelism and memory locality in a program.

3.1.1 Dependences in Program

A *dependence* between any two statements in a program is defined as a relation that constraints their execution order. There are two types of dependences which can occur in a program. *Control dependence* is a constraint that arises due to the control flow of a program whereas, a *data dependence* arises from the flow of data between statements. Dependence is a principal tool used by compilers in analyzing and transforming programs for their execution on parallel and vector machines. It is helpful in determining when it is safe to make certain program transformations. A transformation is "safe", if the transformed program preserves the correctness of the input program.

3.1.2 Validity or Legality of a Transformation

Two computations are equivalent if given the same inputs, they produce identical values for output variables. A valid transformation preserves all dependences in a program. In other words, any transformation that reorders the execution of statements in a program is said to be valid/legal if the transformation maintains the order of source and sink for every dependence present in the program.

3.1.3 Formal Definitions

Some of the important terms which are used frequently throughout the chapters are enumerated below. These terms are well defined in the literature [10].

3.1.3.1 Data Dependence

There is a *data dependence* from statement s_1 to statement s_2 (statement s_2 dependes on statement s_1) if and only if

1. both statements access the same memory location and at least one of them stores into it and

2. there is a feasible run-time execution path from s_1 to s_2.

3.1.3.2 Iteration Number

For an arbitrary loop in which the loop index I runs from L and U in steps of S, the (normalized) *iteration number* i of a specific iteration is equal to the value $(I - L + 1)/S$, where I is the value of the index on that iteration.

3.1.3.3 Nesting Level

In a loop nest, the *nesting level* of a specific loop is equal to one more than the number of loops that enclose it.

3.1.3.4 Perfect loop nest, Imperfect loop nest

A set of nested loops is called a perfect loop nest iff all statements appearing in the nest appear inside the body of the innermost loop. Otherwise, the loop nest is called an imperfect loop nest.

3.1.3.5 Loop Dependence

There exists a dependence from statement s_1 to statement s_2 in a common nest of loops if and only if there exisis two iteration numbers i and j for the nest such that

1. $i < j$ or $i = j$ and there is a path from s_1 to s_2 in the body of the loop
2. statement s_1 accesses memory location M on iteration i and statement s_2 accesses location M on iteration j and
3. one of these accesses is a write.

3.1.3.6 Dependence Distance Vector

Suppose that there is a dependence from statement s_1 on iteration i of a loop nest and statement s_2 on iteration j then the *dependence distance vector* $d(i,j)$ is defined as a vector of length n such that $d(i,j)_k = j_k - i_k$.

3.1.3.7 Dependence Direction Vector

Suppose that there is a dependence from statement s_1 on iteration i of a loop nest of n loops and statement s_2 on iteration j, then the *dependence direction vector* is $D(i,j)$ is defined as a vector of length n such that $D(i,j)_k = \begin{cases} " < " & if\ d(i,j)_k > 0 \\ " = " & if\ d(i,j)_k = 0 \\ " > " & if\ d(i,j)_k < 0 \end{cases}$

3.1.3.8 Direction Matrix

The *direction matrix* for a nest of loops is a matrix in which each row is a direction vector for some dependence between statements contained in the nest and every such direction vector is represented by a row.

3.1.3.9 Loop-carried Dependence

Loop-carried dependence, is the dependence which exists only when the loop is iterated. Statement s_2 is said to have a loop-carried dependence on statement s_1 if and only if s_1 references location M on iteration i, s_2 references M on iteration j and $d(i,j) > 0$ (that is, $D(i,j)$ contains a '<' as leftmost non '=' component).

3.1.3.10 Loop-independent Dependence

Loop-independent dependence is the dependence which exists because of the position of the code within the loops. Statement $s2$ has a loop-independent dependence on statement $s1$ if and only if there exist two iteration vectors i and j such that:

1. Statement $s1$ refers to memory location M on iteration i, $s2$ refers to M on iteration j, and $i = j$.

2. There is a control flow path from $s1$ to $s2$ within the iteration.

3.1.3.11 Reordering Transformation

A *reordering transformation* is a program transformation that merely changes the order of execution of the code, without adding or deleting any execution of any statement.

3.1.4 Validity of a Transformation in terms of Dependence Direction Vector

Let T be a transformation which when applied to a loop nest, does not rearrange statements in the body of the loop. Then, T is said to be valid if it does not result in dependences direction vectors having a leftmost non '=' component that is '>'.

3.2 Some Parallelizing Transformations

Some of the parallelizing transformations described in the literature [10] and [8] are explained below.

3.2.1 Loop Interchange

Loop interchange is a reordering transformation that swaps the ordering of two adjacent loops in a perfect loop nest. This transformation is useful in extracting parallelism. It is desirable to move loops with no dependences to outermost possible

position. This shifting makes parallelization of outer loops feasible. Reordering of a large number of statements takes place with this single transformation. So, loop interchange is an important transformation.

Loop Interchange is legal if and only if any two statements in the inner loop do not contain flow dependence with direction vector $(<, >)$.

3.2.2 Loop Permutation

Loop permutation generalizes loop interchange by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

A permutation of loops in a perfect loop nest is legal if and only if on applying the same permutaion to columns of the direction matrix, the matrix does not have '>' direction as the leftmost non-'=' direction in any row.

3.2.3 Loop Distribution

Loop distribution takes a loop that contains multiple statements and splits it into two loops with the same iteration-space traversal, such that the first loop contains some of the statements from the original loop and the second contains remaining. The transformation eliminates loop carried dependences. So, it can be used to convert a sequential loop to multiple parallel loops.

Loop distribution is legal if it does not result in breaking any cycles in dependence graph of the original loop.

3.2.4 Loop Fusion

Loop fusion takes two adjacent loops that have the same iteration-space traversal and combines their bodies into a single loop.

A fusion of two loops is legal if they have the same bounds and if there are no dependences in the fused loop for which instructions from the first loop depend on instructions from the second loop in the reverse direction.

3.2.5 Loop Reversal

Loop reversal reverses the order in which a particular loop's iterations are performed.

3.2.6 Unroll-and-jam

An *unroll-and-jam* to factor n consists of unrolling the outer loop $n-1$ times to create n copies of the inner loop and fusing those copies together. The transformation improves the efficiency of pipelined functional units.

An unroll-and-jam to factor n is legal if and only if there exists no dependence with direction vector $(<,>)$, such that the dependence distance for outer loop is $< n$.

3.3 ROSE Compiler Infrastructure

ROSE [23], [24], [25] is an open source compiler infrastructure. It is developed by *Lawrence Livermore National Laboratory(LLNL)*. It provides support for multiple languages. It can read and analyze source code written in the languages like C, C++ and Fortran. It is useful for building tools for the program analysis and transformation as well as code generation. ROSE is designed to build translators. It uses source-to-source approach to define such translators. It provides tools to support the program analysis. It also provides support for users to build their own forms of analysis and specialized transformations.

ROSE infrastructure works by reading the source code and generating an Abstract Syntax Tree (AST). The source-to-source translator takes the source code written in the high level language, as input, generates AST, performs the various operations, transforms the AST again into High level language, and then outputs the transformed code. The nodes used to define AST graph are Intermediate Representation (IR). ROSE provides mechanisms to traverse and manipulate the AST and also to regenerate source code from the AST.

Chapter 4

Specification of Transformations

Some parallelizing transformations were discussed in section 3.2. While the transformations considered can be expressed in terms of dependence relations, they need to be formalized to be consumed by the software. We need a formal specification method which is powerful enough to express all the parallelizing transformations. Such a formalization is beneficial to understand the properties, proper ordering and the effects of application of the transformations.

Each transformation is specified in the form of a precondition and an action part. The precondition part specifies the legality condition for the application of the transformation so that the transformation does not change the semantics of the program. The action part is a set of primitive actions which give the steps to perform the transformation.

This chapter gives an overview of the method that we have used to specify parallelizing transformations. Section 4.1 describes the formal method similar to predicate logic. The formalization is used for the theoretical understanding of the parallel transformations. Section 4.2 describes the specification language used by our system. It explains some of the important language constructs and action routines of the language.

4.1 A Formal Specification Method

The specifications are expressed in a notation similar to the predicate form (Q:R:P) read as, *Q such that R for which P is true*, where Q is a set of quantifiers, R is a valid formula and P is a predicate. A nested specification is possible for complex transformation as the predicate P is also expressed in the same (Q:R:P) format, including quantifiers, a formula and the predicates. In a simple case, predicate can take a form of a valid formula. This method is given by Paleri, V. [4] for the formal specification. The method is powerful enough to express many complex transformations. We have used this formalization method to express parallelizing optimizations. This helps to understand the properties of transformations. The transformations are then modeled into the specification language.

Following are the examples of parallel transformations specified in the (Q:R:P) notation.

4.1.1 Formal Specification: Loop Interchange

Let us assume, we have a perfectly nested loop, say L. Let L_1 and L_2, are outer and inner loops respectively in the loop nest L. The loop header of any loop, say l is denoted as *l.head*. It consists of the initialization statement(s), the condition statement(s), and the increment/decrement statement(s). Then, the preconditions and the action corresponding to the loop interchange are as follows.

Precondition:

1. Flow dependence should not exist from $L.L_1$.head to $L.L_2$.head.

$$\neg\ (L.L_1.head\ \delta\ L.L_2.head)$$

2. There should not exist a flow dependence with direction vector $(<,>)$ between any two statements of the loop nest L.

$$\forall S_i \forall S_j : (S_i \in L) : \neg(S_i \delta_{(<,>)} S_j)$$

Action: If the preconditions are satisfied, swap the headers of the two loops L_1 and L_2, including the initialization statement, condition statement and the increment/decrement statement.

$$\{\ swap(L.L_1.head, L.L_2.head)\ \}$$

The (Q:R:P) notation

$\forall(L)$:

$\neg\ (L.L_1.head\ \delta\ L.L_2.head)$:

$(\forall S_i \forall S_j : (S_i \in L) \wedge (S_j \in L) : \neg(S_i \delta_{(<,>)} S_j))$

$\{\ swap(L.L_1.head, L.L_2.head)\ \}$

The above notation is an example of nested specification. Here, precondition is also expressed in the same (Q:R:P) format.

4.1.2 Formal Specification: Loop Fusion

Let us assume we have two loops, say L_1 and L_2, such that L_1 is executing before L_2. Then, the precondition and action for the application of Loop Fusion are as follows.

Precondition:

1. The two loops should iterate through the same interval with equal upper and lower bounds.

   $$(L_1.LB == L_2.LB)\ \wedge\ (L_1.UB == L_2.UB)$$

   where,

   LB stands for Lower bound of the loop, and

   UB stands for the Upper bound of the loop.

2. After fusion, anti-dependence should not be introduced from the statements of L_2 to the statements of L_1.

   $$\forall S_i \forall S_j : (S_i \in L_1) \wedge (S_j \in L_2) : \neg(S_j\ \delta_a\ S_i)$$

Action:

Merge the statements of the two loops into one single loop.

$$\{\ merge(L_1, L_2)\ \}$$

The (Q:R:P) notation

$\forall(L_1, L_2)$:

$(\ (L_1.LB == L_2.LB)\ \wedge\ (L_1.UB == L_2.UB))$:

$(\forall S_i \forall S_j : (S_i \in L_1) \wedge (S_j \in L_2) : \neg(S_j\ \delta_a\ S_i))$

$\{\ merge(L_1, L_2)\ \}$

4.2 The Specification Language

The specification language we have used to specify transformations takes the following mentioned points into consideration.

- The specification language should be similar to the (Q:R:P) notation to get the same power of expression. It should be able to express all the traditional parallelizing optimizations. If some new transformation is to be introduced or to be experimiented, it should be easily expressible in the specification language.
- The optimizer generator software expects specification as an input. So, the specifications should be written in a way such that it can be easily understood by the software.
- The aim of the designed system is to reduce the efforts of writing parallel transformations. So, the specification language should be simple and easy to learn for the compiler writers.
- The language should not be very conservative. The user should be able to express optimizations as aggressively as possible.
- The language should be extendable to provide scope of addition of new routines and predicates.

Our specification language is a 'C'-like language which can be easily adopted by any programmer. The language provides *modularity* with the scope of writing one or many 'subspecifications', along with the 'mainspecification' to specify complex transformations. The language is just a sentential modelling of the (Q:R:P) notation which combines *expression power* of the formal notation with the *easy syntax* of the sentential notations. The language offers flexibility in writing preconditions and action routines so that transformations can be specified *aggressively*.

4.2.1 Format of the Specification Language

In this part of the section, the format of the specification in the specification language is explained corresponding to some of the grammar rules in the language.

Grammar rules are shown in figure 4.1

```
Rule 1:: specification : subspecificationList mainspecification

Rule 2:: subSpecification : subSpecificationHeader subSpecificationBody

Rule 3:: subSpecificationHeader : retOrArgType ID  ( formalArguments ) varDeclaration

Rule 4:: mainSpecification : ID ( formalArguments ) varDeclaration mainSpecificationBody

Rule 5:: subSpecificationBody : compoundStatement
         mainSpecificationBody : compoundStatement

Rule 6:: compoundStatement : BEGIN statementList END

Rule 7:: statement : languageConstruct ';'
                   | primitiveAction ';'
                   | RETURN ID ';'
                   | RETURN boolValue ';'
```

Figure 4.1: Rules in the Specification Language

1. *Rule 1:*

   Every specification consists of two parts, an optional list of one or more subspecifications and the mainspecification.

2. *Rule 2:*

   Both the mainspecification and the subspecification contain two parts, header

and the body. A subspecification accepts a list of formal arguments and returns some value. The *subspecification* is similar to a function or a subroutine in 'C'-like languages. This provides modularity in the specification and makes it easy to write complex transformations.

3. *Rule 3:*

   The subspecification header includes retOrArgType i.e. return value, ID i.e. name of the specification, list of formal arguments enclosed within parantheses and variable declaration which declares local variables of the subspecification.

4. *Rule 4:*

   The *mainspecification* consists of the main precondition and action part. Both subspecification and mainspecification define their own local variables. The header of the main specification consists of ID i.e. name of the specification, a list of formal arguments and the variable declaration part to declare local variables.

5. *Rule 5:*

   The body of the subspecification and the mainspecification consist of a statement or a compound statement.

6. *Rule 6:*

   In a compound statement, the statements are enclosed within the token *begin* and the token *end*.

7. *Rule 7:*

   A statement consists of a *language construct*, a *primitive action* or a *return statement*. The language construct includes *forall* statement or an *if* statement. The primitive actions are basic functions required to be performed by the transformation.

All the rules explained in this section consist of the basic constructs that are used in general, in any specification. However, in case of parallelizing transformations,

mostly loop dependences are taken into consideration. Some of the language constructs and primitive actions provided for dependence checking and to manipulate loop structures are given below.

- *FLOWDEP*: To check flow dependence between the two statements
- *FLOWDEPDIR*: To check the direction of the flow dependence.
- *ANTIDEP*: To check the existence of anti-dependence between the statements.
- *INLOOP*: To check whether a statement is present inside a loop or not.
- *loop createNewLoop(head)*: To create a new loop with the given header.
- *addStatementToLoop(loop L, statement S)* and *deleteStatementFromLoop(loop L, statement S)*: To add a statement to a given loop and to delete a statement from a loop.
- *bool flowDepDirMat(statement S1, statement S2, int pos1, int pos2, char dir1, char dir2)*: returns true if the direction vector for the dependence between statements S1 and S2 has the directions dir1 and dir2 at positions pos1 and pos2 respectively.

So, the description given above is an overview of the specification language we have used. There are many other constructs provided by the language. Some new rules can be easily added to the language to provide some new features, if required. Loop Interchange and Loop Fusion are specified in the specification language as below.

4.2.2 Specification: Loop Interchange

Figure 4.2 shows the specification for Loop Interchange.

4.2.3 Specification: Loop Fusion

Figure 4.2 shows the specification for Fusion.

```
boolean legalINX (nestedforloop L)
var
      statement SI, SJ;
begin
forall (SI) satisfy ( inLoop ( L , SI ))
begin
      if exists (SJ) satisfy (inLoop (L, SJ) and (flowDep (SI, SJ) and flowDepDir (SI, SJ, <, >)))
            return false;
      ;
end
;
return true;
end

loopInterchange
var
      nestedforloop L;
begin
forall (L) satisfy (notflowDep (L.L1.head, L.L2.head))
      if legalINX (L)
            swap(L.L1, L.L2);
      ;
;
end
```

Figure 4.2: Loop Interchange

```
boolean legalFuse(forloop L)
var
    statement SI, SJ;
begin
    forall (SI) satisfy ( inLoop ( L , SI ))
    begin
        if exists (SJ) satisfy (inLoop (L, SJ) and antiDep (SJ, SI))
            return false;
        ;
    end
    ;
    return true;
end

loopFusion
var
    consecutiveforloop cL;
    forloop L;
    statement S;
begin
    forall (cL) satisfy (equals(cL.L1.LB, cL.L2.LB) and equals(cL.L1.UB, cL.L2.UB))
    begin
        L = createNewLoop(cL.L1.head);
        forall(S) satisfy(inLoop (cL.L1, S) or inLoop(cL.L2, S))
        begin
            addStatementToLoop(L, S);
        end
        ;
        if(legalFuse(L))
            replace(cL, L);
    end
    ;
end
```

Figure 4.3: Loop Fusion

Chapter 5

Generating Parallel Transformers

We have designed and developed a system which helps in generating parallel transformers. The system provides a framework which accepts a specification as an input, parses it, and generates code for it. The generated code is then applied on a C or C++ program to get the transformed code. We have written specifications for some of the important parallel transformations, and generated code for them. Our system can either be used as a tool which helps in generating code transformers or as an environment for experimenting with code transformations. This chapter gives a brief overview of the framework that we have developed.

5.1 The Overview of the framework

The components of the system shown in figure 5.1 are described below.

An **HLL source code** is an input program written in high level language. The **Parser** converts it to the **Intermediate code**. ROSE uses Abstract Syntax Tree (AST) for Intermediate representation. Parallelizing transformations involve loop operations such as modification of loop structures, recomputation of loop bounds, or renaming of loop index variables. So, a high level Intermediate Representation that retains information about loop structures from the source program is needed. In AST, every node represents an IR for a specific code element. Hence, the use of AST for intermediate representation has a significance for our framework.

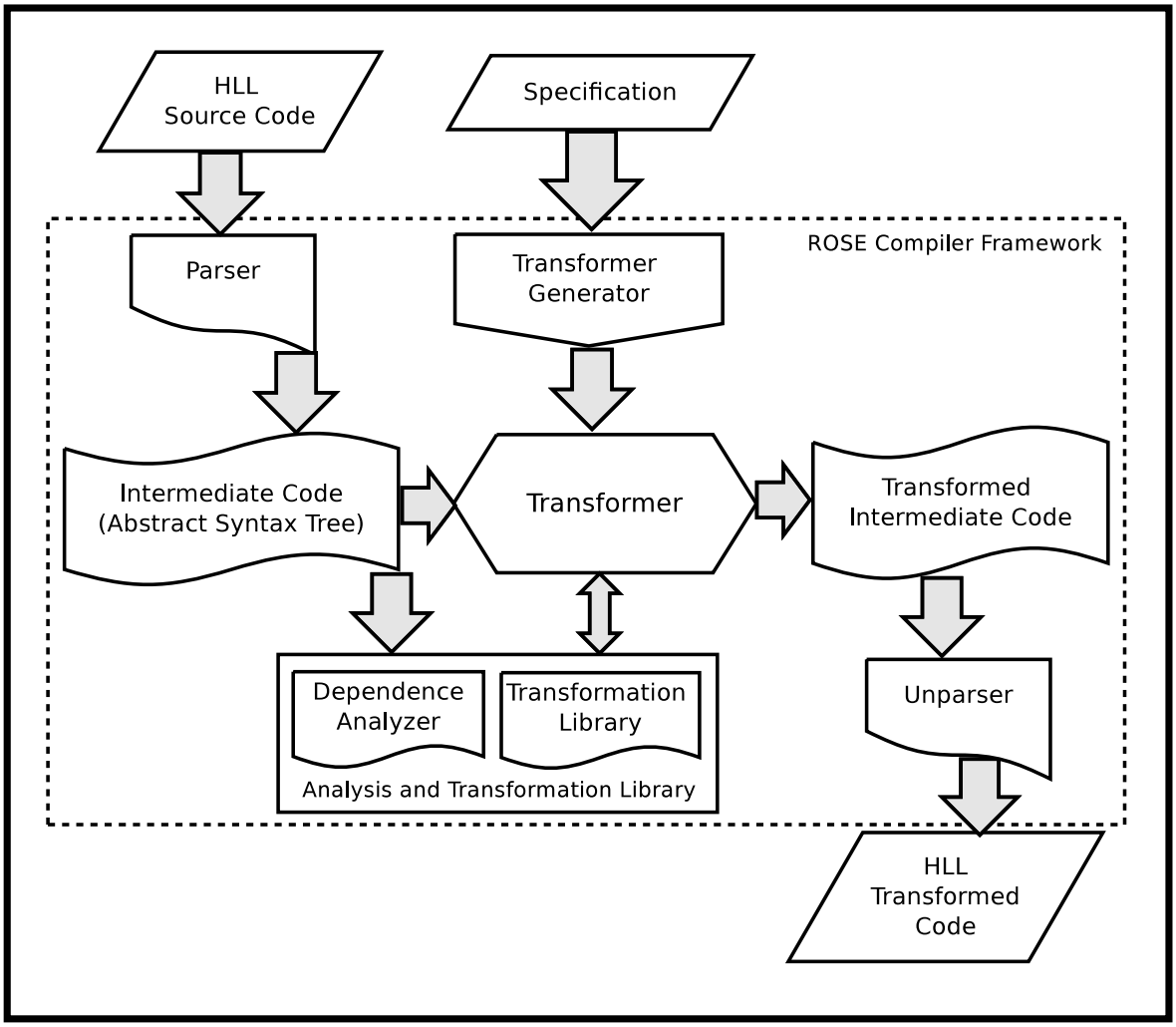

Figure 5.1: The transformer Generator Framework

The Analysis and Transformation Library contains some important implemented methods to assist the transformation process. It has two components as follows.

- *The Dependence Analyzer* performs dependence analysis on the IR of the source code and returns the corresponing dependence matrix. It uses methods from ROSE Compiler to build dependence graph and to generate dependence direction vectors for each dependence relation.

- *The Transformation Library* contains the implementations for the primitives provided by the specification language. These methods are called by the transformer during the transformation process. The methods are applied to perform various actions on the IR of the input program. The transformation library uses ROSE methods to access and manipulate AST.

The **Transformer Generator** is the most important processing unit in the framework. It takes **Specification** as an input and generates **Transformer** from it. Section 5.2 describes the functionality of the transformer generator tool. The transformer performs specified transformations on the IR of the input program and generates the corresponding transformed IR. It contains functions corresponding to the *subspecifications* and *mainspecification* specified in the given specification. Transformer uses analysis and transformation library in the transformation process. To check the preconditions specified in the specification, the transformer uses the dependence matrix generated by the dependence analyzer. To perform the specified action routines, the transformer calls the methods implemented in the transformation library.

The **Unparser** takes the **transformed IR** as an input and converts it to the **High Level Language Transformed Code** which is the final output of the system.

5.2 The Transformer Generator

The transformer generator takes specifications for code transformations as an input. These specifications are written in the specification language. For the generation of the transformer, the transformer generator uses the method of syntax-directed translation. For every production rule in the grammar, some semantic actions are defined to generate code for the transformer. The tool uses the following steps to generate transformers:

1. It performs *Lexical and Syntax Analysis of the specification* using Lex and Yacc.

2. It generates code to call the *parser* which converts the input program into the corresponding intermediate representation.

3. It generates code for the *identification of code elements* which are of interest. The generated code iterates to traverse the whole AST to search for the required code elements and creates a list of the code elements.

4. It generates code to call dependence analyzer, to *collect the dependence information* of all the dependence relations present in the input program and to create a dependence matrix to store the collected information.

5. It generates code which uses the dependence information of the program and *checks for the precondition* specified in the specification.

6. It generates code to call the primitives implemented in the transformation library and *perform action routines* specified in the specification of the transformer.

7. Finally, it invokes the *unparser* which gives the transformed high level language code.

5.3 Example Transformer: Loop Interchange

As an example, figure 5.2 shows the generated transformer code for the subspecification of loop interchange. The commented code is added manually to show the parts of the specification corresponding to which transformer code is generated.

```
//boolean legalINX (nestedforloop L)
bool ROSE_legalINX (SgNestedForStatement *L)
{
//var
//      statement SI, SJ;
        SgStatement *SI, *SJ;

//forall (SI) satisfy ( inLoop ( L.L2 , SI ))
        Rose_STL_Container<SgNode*> stmtList = NodeQuery::querySubTree (L->L1, V_SgStatement);
        Rose_STL_Container<SgNode*>::iterator si = stmtList.begin();
        for (; si!= stmtList.end(); si++ )
        {
                SI = isSgStatement(*si);
                if(ROSE_inLoop(L->L2, SI))
                {

        //if exists (SJ) satisfy (inLoop (L.L2, SJ) and (flowDep (SI, SJ) and flowDepDir (SI, SJ, <, >)))
                        Rose_STL_Container<SgNode*>::iterator sj = stmtList.begin();
                        for (sj=si+1; sj!= stmtList.end(); sj++ )
                        {
                                SJ = isSgStatement(*sj);
                                if(ROSE_inLoop(L->L2, SJ) && (ROSE_flowDep(SI, SJ) && ROSE_flowDepDir(SI, SJ, '<', '>')))
                                        return false;
                        }
                }
        }
        return true;
}
```

Figure 5.2: Loop Interchange Subspecification: Generated Transformer

Chapter 6

Experiments and Results

We have written specifications for some parallel transformations and generated the corresponding transformer using our framework. Generated transformers are then applied on various inputs. The inputs we have used are the code snippets satisfying different dependence conditions. These code snippets are taken from the examples given in [10]. The inputs given to every transformer cover almost all the test cases required to test the specific transformation.

6.1 Loop Interchange

For a 2-perfectly nested loop, the application of loop interchange transformation is legal if head of the second loop does not depend on the head of the first loop and the statements inside loop does not have a flow dependence between them with the direction vector $(<, >)$.

6.1.1 Specification: Loop Interchange

Figure 6.1 shows the specification written for Loop Interchange.

```
boolean legalINX (nestedforloop L)
var
      statement SI, SJ;
begin
forall (SI) satisfy ( inLoop ( L , SI ))
begin
      if exists (SJ) satisfy (inLoop (L, SJ) and (flowDep (SI, SJ) and flowDepDir (SI, SJ, <, >)))
            return false;
      ;
end
;
return true;
end

loopInterchange
var
      nestedforloop L;
begin
forall (L) satisfy (notflowDep (L.L1.head, L.L2.head))
      if legalINX (L)
            swap(L.L1, L.L2);
      ;
;
end
```

Figure 6.1: Specification: Loop Interchange

6.1.2 Test Cases: Loop Interchange

We need a perfectly nested loop consisting of two loops to perform Loop Interchange. We have considered different cases where statements in the loop have different direction vectors.

1. ***Case 1: Dependence Vector*** $(=, =)$

   Case 1 considers a loop with loop-independent dependence i.e. dependence with the direction vector as $(=, =)$. Figure 6.2 shows the input given and output obtained for this case. Loop Interchange is legal in this case. So, the output shows the interchanged loops.

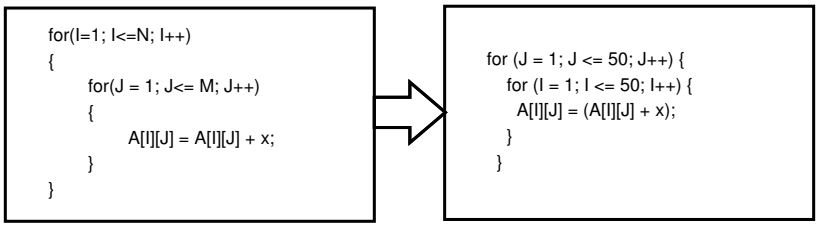

Figure 6.2: Loop Interchange: Case 1

2. ***Case 2: Dependence Vector*** $(=, <)$

   Case 2 takes care of the loop which has loop-carried dependence with direction

vector as $(=, <)$. Figure 6.3 shows the input and output for this case. Loop Interchange transformation is performed as it is legal in this case.

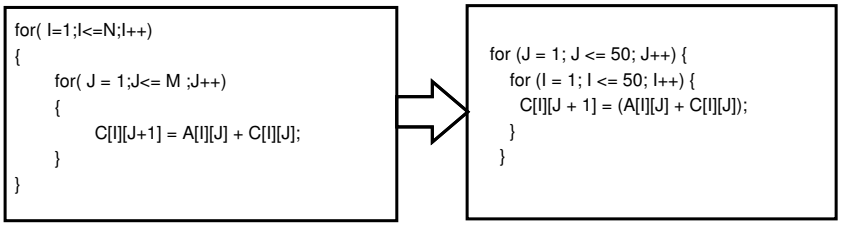

Figure 6.3: Loop Interchange: Case 2

3. ***Case 3: Dependence Vector*** $(<, =)$

In case 3, the loop shows a loop-carried dependence with direction vector as $(<, =)$. A legal loop interchange is performed with the loops as shown in the figure 6.4

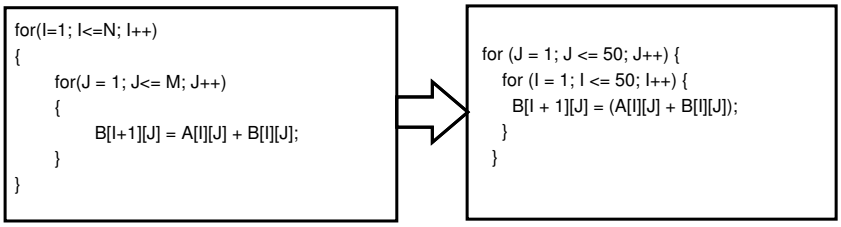

Figure 6.4: Loop Interchange: Case 3

4. ***Case 4: Dependence Vector*** $(<, <)$

Case 4 is an example of dependence carrying loop with direction vector, $(<, <)$. It is a candidate for valid loop interchange transformation. Figure 6.5 shows the corresponding input ans output.

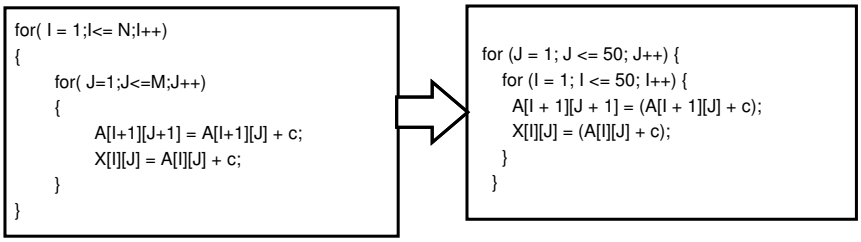

Figure 6.5: Loop Interchange: Case 4

5. ***Case 5: Dependence Vector*** $(<, >)$

Case 5 considers a case where loop transformation is not legal. It contains a

loop with the statements inside the loop having dependence vector as $(<, >)$. Figure 6.6 shows the input and output for this case.

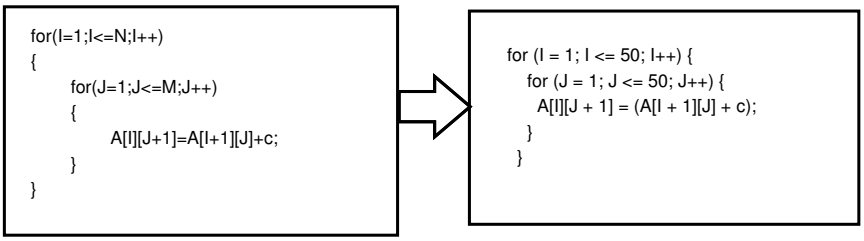

Figure 6.6: Loop Interchange: Case 5

6. ***Case 6:*** $(L_1.head\ \delta\ L_2.head)$

   In this case, as shown in figure 6.7, the head of the second loop is dependent on the head of the first loop. This condition restricts loop interchange transformation.

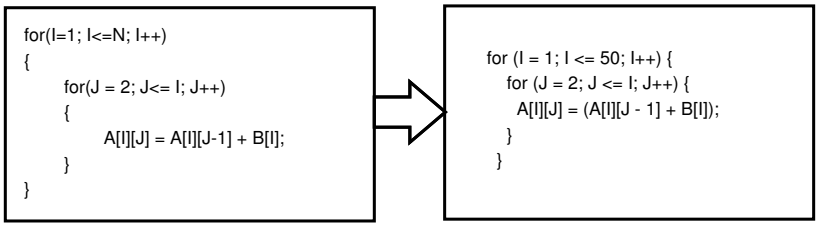

Figure 6.7: Loop Interchange: Case 6

6.2 Loop Distribution

Every statement in a loop can be legally distributed if it does not contain any loop carried backward anti-dependence. We have considered two cases to specify loop distribution. In the first case, if the precondition is satisfied, every statement in the loop is executed in different loop resulting in consecutive loop sequence with number of loops equals the number of statements in the original loop. In the second case, when the precondition is not satisfied, only the statements which are not dependent on any other statements are distributed into the new loops. In this case, the number of new loops equals the number of loop-independent statements in the original loop.

6.2.1 Specification: Loop Distribution

Figure 6.8 shows the specification written for Loop Distribution.

```
consecutiveforloop distributeAll(forloop L)
var
      statement S;
      forloop L1;
      consecutiveforloop C;
begin
      forall(S) satisfy(inLoop(L, S))
      begin
            L1 = createNewLoop(L.head);
            addStatementToLoop(L1, S);
            deleteStatementFromLoop(L, S);
            addLoop(C, L1);
      end;
      return C;
end

consecutiveforloop distributeIndependent(forloop L)
var
      statement SI, SJ;
      forloop L1;
      consecutiveforloop C;
begin
      forall(SI) satisfy(inLoop(L, SI))
      begin
            if not exists(SJ) satisfy(flowDep(SI, SJ) or antiDep(SJ, SI))
                  L1 = createNewLoop(L.head);
                  addStatementToLoop(L1, S);
                  deleteStatementFromLoop(L, S);
                  addLoop(C, L1);
            ;
      end;
      addLoop(C, L);
      return C;
end
```

```
boolean legalDistribute (forloop L)
var
      statement SI, SJ;
begin
forall (SI) satisfy ( inLoop ( L , SI ))
begin
 if exists (SJ) satisfy (inLoop (L, SJ) and antiDep (SJ, SI))
            return false;
      ;
end
;
return true;
end

loopDistribution
var
 forloop L;
      consecutiveforloop cL;
begin
      forall (L)
       if legalDistribute (L)
        cL = distributeAll(L);
            else
                  cL = distributeIndependent(L);
       ;
      ;
      replace(L, cL);
end
```

Figure 6.8: Specification: Loop Distribution

6.2.2 Test Cases: Loop Distribution

We have shown here two test cases, one shows legal transformation of loop distribution and the other shows the unchanged loop when the precondition is not satisfied and it does not contain any loop independent statement.

1. ***Case 1: Legal Loop Distribution***

   Case 1 considers a loop with the statements not having anti-dependence between them. Figure 6.9 shows the input given and the transformed output.

2. ***Case 2: Invalid Loop Distribution***

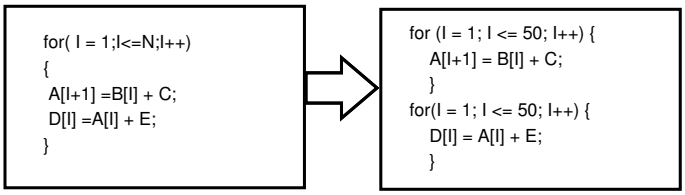

Figure 6.9: Loop Distribution: Case 1

Case 2 shows a loop where statements have backward anti-dependence between them. So, the transformation is invalid. Figure 6.10 shows the input and unchanged code in the output.

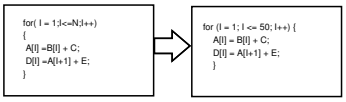

Figure 6.10: Loop Distribution: Case 2

6.3 Loop Fusion

A loop fusion is legal if it does not introduce any loop carried backward dependence in the merged loop. To check this condition, first the two loops are merged into a new loop. If the new loop does not contain any anti-dependence, the original consecutive loops are replaced with the new merged loop.

6.3.1 Specification: Loop Fusion

Figure 6.11 shows the specification written for Loop Fusion.

```
boolean legalFuse(forloop L)
var
    statement SI, SJ;
begin
    forall (SI) satisfy ( inLoop ( L , SI ))
    begin
        if exists (SJ) satisfy (inLoop (L, SJ) and antiDep (SJ, SI))
            return false;
        ;
    end
    ;
    return true;
end

loopFusion
var
    consecutiveforloop cL;
    forloop L;
    statement S;
begin
    forall (cL) satisfy (equals(cL.L1.LB, cL.L2.LB) and equals(cL.L1.UB, cL.L2.UB))
    begin
        L = createNewLoop(cL.L1.head);
        forall(S) satisfy(inLoop (cL.L1, S) or inLoop(cL.L2, S))
        begin
            addStatementToLoop(L, S);
        end
        ;
        if(legalFuse(L))
            replace(cL, L);
    end
    ;
end
```

Figure 6.11: Specification: Loop Fusion

6.3.2 Test Cases: Loop Fusion

We have shown here two test cases, one shows legal transformation of loop fusion and the other shows the unchanged loop when the precondition is not satisfied and the application of the transformation is invalid.

1. ***Case 1: Valid Loop Fusion***

   Case 1 considers a loop with the statements not having anti-dependence between them in the merged loop. Figure 6.12 shows the input given and the transformed output.

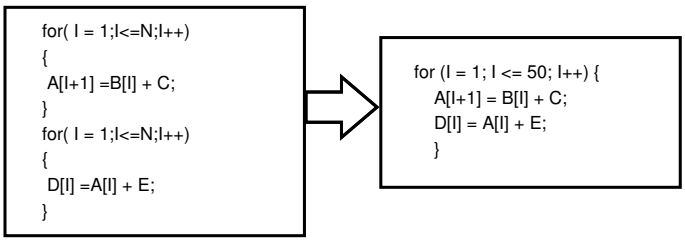

Figure 6.12: Loop Fusion: Case 1

2. ***Case 2: Invalid Loop Fusion***

   Case 2 shows a loop where backward anti-dependence is introduced after merging the loops. So, the transformation is invalid. Figure 6.13 shows the input and unchanged code in the output.

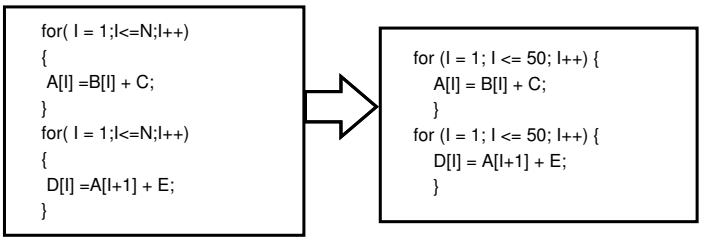

Figure 6.13: Loop Fusion: Case 2

6.4 Loop Reversal

Loop Reversal is a simple transformation of reversing the direction of the iteration space. It is always legal to perform this transformation. So, no precondition is

involved to check the legality of the transformation. The transformation consists of a set of action routines to modify initial statement, conditional statement and the increment or decrement statement of the loop head. We have assumed that the loop has only one initial statement, one conditional statement and one increment or decrement statement.

6.4.1 Specification: Loop Reversal

Figure 6.14 shows the specification for Loop Reversal.

```
loopReverse
var
    forloop L;
    int lb, ub;
begin
    forall (L)
    begin
        lb = getInitialValue(L.head);
        ub = getConditionalValue(L.head);
        setInitialValue(L.head, ub);
        setConditionalValue(L.head, lb);
        reverseCondition(L.head);
        if(isIncrement(L.head))
            setDecrement(L.head);
        ;
        if(isDecrement(L.head))
            setIncrement(L.head);
        ;
    end
    ;
end
```

Figure 6.14: Specification: Loop Reversal

6.4.2 Test Cases: Loop reversal

As no preconditions are involved, we have considered only one case here. Figure 6.15 shows the input and output for the transformation.

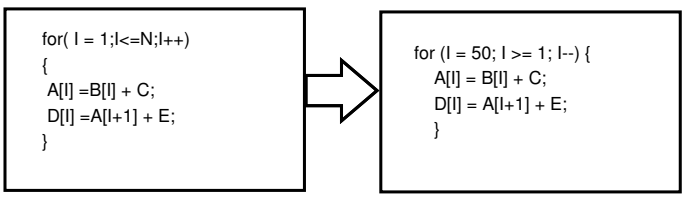

Figure 6.15: Loop Reversal

6.5 Loop Permutation

This is the general form of loop interchange. It swaps two loops at any given position in the perfect loop nest with arbitrary number of loops. A loop permutation is legal if after permutation, first non-'=' member in the dependence direction matrix is '<'. It has a complex precondition depending on the position of the loops to be swapped.

6.5.1 Specification: Loop Permutation

Figure 6.16 gives the specification for loop permutation.

```
boolean legalPermute (nestedforloop L, int d1, int d2)
var
      statement SI, SJ;
begin
      forall (SI) satisfy ( inLoop ( L.nest[last] , SI ))
      begin
      if exists(SJ) satisfy(inLoop(L.nest[last], SJ) and flowDep(SI,SJ) and
(flowDepDirMat(SI, SJ, d1, d2, =, >) or flowDepDirMat(SI, SJ, d1, d2, <, =) or flowDepDirMat(SI, SJ, d1, d2, <, <) )
and GTE(flowDepDirMatFirstLT(SI, SJ),  d1))
            return false;
      end
      ;
      return true;
end

loopPermutation (int l1, int l2)
var
      nestedforloop L;
begin
      forall (L) satisfy (notflowDep (L.nest[l1].head, L.nest[l2].head) )
      if (legalPermute (L, l1, l2))
            swap(L.nest[l1], L.nest[l2]);
      ;
;
end
```

Figure 6.16: Specification: Loop Permutation

6.5.2 Test Cases: Loop Permutation

This transformation takes positions of loops to be swapped as input.

1. ***Case 1: Direction Matrix not containing '>' direction***

   The loop nest shown in case 1 does not contain the direction '>' in the direction matrix. So, the precondition is true for any input positions. Figure 6.17 shows the input given and the trasformed output.

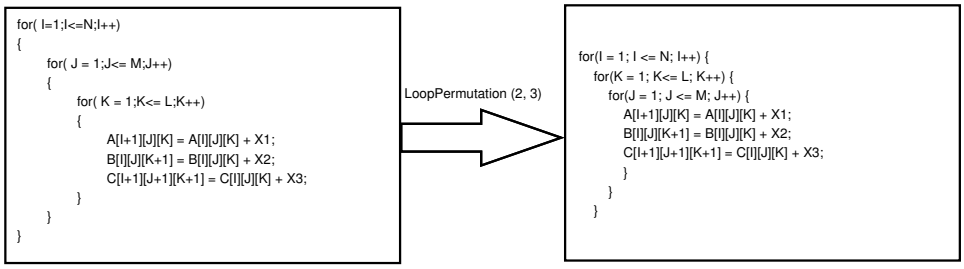

Figure 6.17: Loop Permutation: Case 1

2. ***Case 2: Valid Permutation for Direction Matrix containing '>' direction***

   In this case, direction matrix contains '>' direction but the swapping of two loops at the given position does not make the first non-'='direction in the direction matrix as '>'. So, the transformation is legal as shown in figure 6.18.

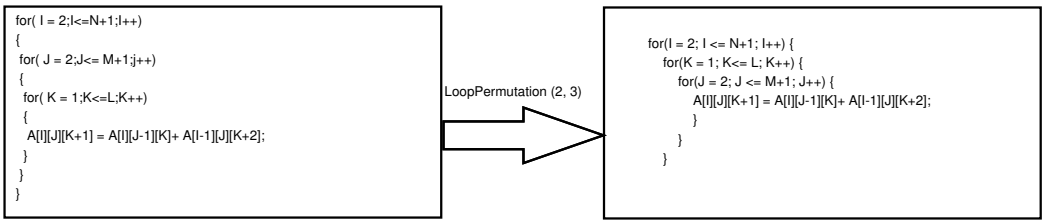

Figure 6.18: Loop Permutation: Case 2

3. ***Case 3: Invalid Permutation for Direction Matrix containing '>' direction***

   In this case, direction matrix contains '>' direction and the swapping of two loops at the given position makes the first non-'='direction in the direction matrix as '>'. So, the transformation is not legal as shown in figure 6.19.

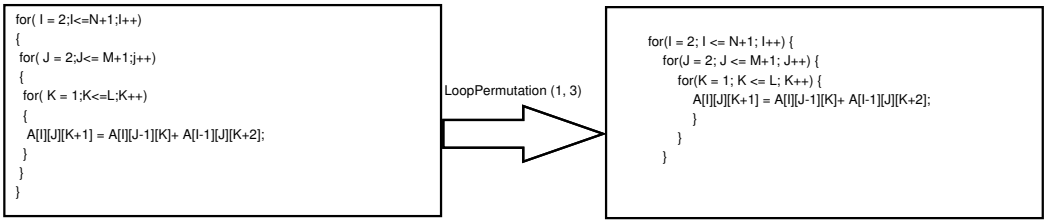

Figure 6.19: Loop Permutation: Case 3

Chapter 7

Conclusion and Future Work

In this thesis work, we have studied different algorithms of parallel transformations to find out the preconditions and action routines associated with them. We specified them formally using the specification language. We have written specifications for some of the traditional parallel transformations. We found that the preconditions in parallel transformations are mostly based on dependence direction vectors and their action routines work with the structures of the loops present in a program. The specification language we have used is derived from the work in [5]. We have added some predefined primitive routines such as *createNewLoop(head)*, *addStatementToLoop(loop L, statement S)*, *depDirMat(statement, statement)*, etc to the language. These routines are useful while writing specifications for parallel transformations.

We have developed a prototype of the framework to generate parallel transformers from the specifications. We have used ROSE Compiler Framework which uses Abstract Syntax Tree (AST) for intermediate representation. The use of AST helps in loop restructuring. Using the framework, we have generated transformers for the specifications we have written. We considered parallel transformations, *Loop Interchange*, *Loop Distribution*, *Loop Fusion*, *Loop Reversal*, and *Loop Permutation* for experimentation. The transformers generated from the specifications are applied on different input programs written in C language. These inputs are the code snippets from [10] and satisfy different dependence conditions.

We have generated transformers as aggressive as possible but in some cases, it becomes conservative. For example, in case of loop distribution and loop fusion, specifications become conservative if anti-dependences are encountered. This leaves a scope for improvement in specifications. **Addition of more primitives** to the specification language can help to overcome this issue.

The framework developed in this thesis provides legality check for the transformations so that the semantics of the programs remain valid. A parallel transformer turns out to be beneficial if after its application, the parallelism in the program increases. However, it is not always beneficial to apply the parallel transformations. In some cases, transformed loops becomes parallelizable while in some cases, parallelizable loops are transformed into non-parallelizable form. For the selective transformation, which loops are to be transformed, **cost based analysis** is required. This can be an important future work associated with this thesis.

The framework aims to increase number of parallelizable loops from the input program. To make them run in parallel, some **parallelization technique** is needed. So, associating some automatic parallelization technique with this work can be an another important extension.

Bibliography

[1] Deborah Whitfield and Mary Lou Soffa. Automatic generation of global optimizers. In *ACM SIGPLAN Notices*, volume 26, pages 120–129. ACM, 1991.

[2] Deborah Whitfield and Mary Lou Soffa. The design and implementation of genesis. *Software: Practice and Experience*, 24(3):307–325, 1994.

[3] Deborah L Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, 1997.

[4] Vineeth Kumar Paleri. *An Environment for automatic generation of code optimizers*. PhD thesis, 1999.

[5] Kingshuk Karuri. A framework for automatic generation of code optimizers, 2001.

[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[7] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[9] Utpal K Banerjee. *Dependence analysis for supercomputing*. Kluwer Academic Publishers, 1988.

[10] Randy Allen. Optimizing compilers for modern architectures: A dependence-based approach author: Randy allen, ken kennedy, publisher: Mo. 2001.

[11] Duo Liu, Yi Wang, Zili Shao, Minyi Guo, and Jingling Xue. Optimally maximizing iteration-level loop parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):564–572, 2012.

[12] Zhao Jie, Zhao Rongcai, and Yao Yuan. A nested loop fusion algorithm based on cost analysis. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1096–1101. IEEE, 2012.

[13] Cosmin E Oancea and Lawrence Rauchwerger. Logical inference techniques for loop parallelization. In *ACM SIGPLAN Notices*, volume 47, pages 509–520. ACM, 2012.

[14] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, P Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *ACM SIGPLAN Notices*, volume 46, pages 549–562. ACM, 2011.

[15] Yong hun Eom, Stephen Yang, James C Jenista, and Brian Demsky. Doj: dynamically parallelizing object-oriented programs. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[16] Albert Noll and Thomas R Gross. An infrastructure for dynamic optimization of parallel programs. *ACM SIGPLAN Notices*, 47(8):325–326, 2012.

[17] Charles Farnum. Dora-an environment for experimenting with compiler optimizers. In *Computer Languages, 1992., Proceedings of the 1992 International Conference on*, pages 86–95. IEEE, 1992.

[18] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[19] Joachim Thees and Reinhard Gotzhein. The experimental estelle compiler: automatic generation of implementations from formal specifications. In *Proceedings of the second workshop on Formal methods in software practice*, pages 54–61. ACM, 1998.

[20] Jack W Davidson and Christopher W Fraser. Automatic generation of peephole optimizations. *ACM SIGPLAN Notices*, 39(4):104–111, 2004.

[21] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ACM Sigplan Notices*, volume 41, pages 394–403. ACM, 2006.

[22] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *ACM Sigplan Notices*, volume 45, pages 389–402. ACM, 2010.

[23] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, volume 2011, page 1, 2011.

[24] Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. Rose user manual: A tool for building source-to-source translators draft user manual (version 0.9. 5a). 2011.

[25] D Quinlan, M Schordan, R Vuduc, T Panas Q Yi, C Liao, and JJ Will-cock. Rose tutorial: A tool for building source-to-source translators.