ACKNOWLEDGEMENT

I would like to thank my thesis supervisor Assoc. Prof. Dr. Haluk Topçuoğlu for his guidance and his unvaluable support during my graduate study. Would like to thank Assoc. Prof. Dr. Mahmut Kandemir (from Pennslyvania State University) for his feedback and support.

I am also profoundly thankful to my husband and family for their patient and encourangement not only during my M.S. stufy but also all through my life.

TABLE OF CONTENT

PAGE NO

ACKNOWLEDGEMENT i

TABLE OF CONTENT ii

ÖZET v

ABSTRACT vi

LIST OF SYMBOLS / ABBREVIATIONS vii

LIST OF FIGURES viii

LIST OF TABLES xi

PART I. INTRODUCTION 1

I.1. OUTLINE OF THE THESIS 3

PART II. COMPILER OPTIMIZATION TECHNIQUES 5

II.1. CACHE RELATED TERMINOLOGY 5

II.2. LOOP TRANSFORMATION TECHNIQUES 7

II.3. DATA DEPENDENCE CONCEPT 10

II.4. MEASURING THE LOCALITY ENHANCEMENT 11

II.5. POLYHEDRAL MODEL 12

II.5.1. Program Execution Model by Polyhedral Method 13

II.6. RELATED WORKS 14

II.6.1. Iterative Compilation 14
II.6.2. Generic Polyhedral Methods 15
***1.a. Polyhedral Representation of Iteration Spaces and Mappings*.........17**
***1.b. Polyhedral Representation of the Dependencies* 19**

PART III. HYBRID GENETIC ALGORITHMS 23

III.1. GENETIC ALGORITHMS 23

III.2. MEMETIC ALGORITMS... 26

III.2.1. Structure of Memetic Algorithms 27

PART IV. PROPOSED FRAMEWORK FOR IMPROVING THE LOCALITY IN LOOPS 31

IV.1. DETAILS OF THE FRAMEWORK 31

IV.1.1. Parsing the original code into the tokens 33
IV.1.2. Generating the Statement Structures 34
IV.1.3. Finding Dependencies in the Statements 35
IV.1.4. Determining Reuses 36
IV.1.5. Generating Constraints on Unknown Coefficients of Mappings ...38
IV.1.6. Checking the Legality of Mappings 40
IV.1.7. Generating Transformed Code from Mappings 42

IV.2. HYBRID GENETIC ALGORITHM OF THE FRAMEWORK.. 43

IV.2.1. Solution Representation 43

IV.2.2. Initial Population Generation 44
IV.2.3. Recombination Operators 45
IV.2.4. Local Search 46
IV.2.5. Fitness Function Evaluation 47

PART V. EXPERIMENTAL STUDY 49

V.1. PERFORMANCES 51

IV.1.1. Triangular Solve 51
IV.1.2. Red-Black Gauss-Seidel Relaxation 53
IV.1.3. Jacobi 55
IV.1.2. Cholesky Factorization 57

PART VI. CONCLUSIONS 60

REFERENCES 61

APPENDIX A. FOURIER-MOTZKIN ELIMINATION 64

APPENDIX B. FARKAS LEMMA 67

APPENDIX C. OMEGA CALCULATOR AND PROJECT 70

RESUME 73

ÖZET

KARMA GENETİK ALGORİTMALARI KULLANARAK DERLEYİCİ ENİYİLEME İÇİN VERİ YERELLİĞİNİ İYİLEŞTİRMEK

Belleğin yüksek verimle kullanılması performansa yönelik programlar için çok önemlidir. Programların çalışma performansını belirleyen en önemli faktörlerden biri verilerin yetersiz veri yerelliği nedeni ile ön bellekte bulunamamasından kaynaklanmaktadır. Döngü dönüşümleri tekniği kullanılarak ana belleğe ulaşım sırası değiştirilebilir ve yeterli veri yerelliği sağlanabilir.

Bu tezde, karma genetik algorithmalar kullanılarak derleyici en iyleme için veri yerelliğini iyileştirmeye çalışılmıştır. Tezde sunulan çözümde genetik algoritmalar yerel arama teknikleri ile iyileştirilmeye çalışılmıştır. Sunulan algoritmanın performansı çok iyi bilinen derleyicilerin iyileştirmeleri ile değerlendirme deneyleri kullanılarak yapılmıştır. Bu testlerin sonuçları sunulan yöntemin derleyici iyileştirmesi yapılmadığında orjinal programlara göre başarılı olduğunu göstermiştir. Fakat derleyici iyileştirmesi yapıldığında ise %25 inde iyileştirme sağlanırken geri kalanda yaklaşık aynı derecede iyileştirme olduğunu göstermiştir.

Kasım 2005 **Gülşah Yılmaz**

ABSTRACT

IMPROVING DATA LOCALITY for COMPILER OPTIMIZATION by USING HYBRID GENETIC ALGORITHMS

The efficient use of memory is very important for performance-oriented programs. Cache misses due to inefficient data locality is one of the critical factors that determine the performance of the programs. Loop transformation techniques target to change the order of memory accesses to increase the data locality.

This thesis proposes a compiler optimization framework for improving data locality by using a hybrid genetic algorithm. Local search techniques were applied at various phases of the proposed genetic algorithm to improve the quality of solutions. The performance of proposed algorithm was evaluated using well-known commercial compilers' optimizations by using selected benchmark codes. The results indicate that the transformed codes generated by our algorithm significantly outperform the source codes of the corresponding source codes with respect to execution times. Moreover, the transformed codes give better performance than the related source codes that are compiled with compiler provided optimizations for 25% of the test cases.

November 2005 **Gülşah Yılmaz**

LIST OF SYMBOLS / ABBREVIATIONS

S_n : n^{th} statement in the sample code

F_n : Mapping of nth statement

p_c : Crossover probability

m_c : Mutation probability

TC : Transformed Code

SC : Source Code

GA : Genetic Algorithm

MA : Memetic Algorithm

GNU_NO_TC : Performance of Transformed Code Compiled by GNU while GNU optimizations are turned off

GNU_O_TC : Performance of Transformed Code Compiled by GNU while GNU optimizations are turned on

GNU_NO_SC : Performance of Source Code Compiled by GNU while GNU optimizations are turned off

GNU_O_SC : Performance of Source Code Compiled by GNU while GNU optimizations are turned on

SUN_NO_TC : Performance of Transformed Code Compiled by SUN while SUN optimizations are turned off

SUN_O_TC : Performance of Transformed Code Compiled by SUN while SUN optimizations are turned on

SUN_NO_SC : Performance of Source Code Compiled by SUN while SUN optimizations are turned off

SUN_O_SC : Performance of Source Code Compiled by SUN while SUN optimizations are turned on

LIST OF FIGURES

PAGE NO

Figure II.1: (a) A Perfectly Nested Loop, and (b) An Imperfectly Nested Loop. …… 6

Figure II.2: Example Code for Loop Fusion……………………………………….… 7

Figure II.3: Example Code for Loop Fission………………………………………… 8

Figure II.4: Example Code for Loop Tiling……………………………………….. 8

Figure II.5: Example Code for Loop Unrolling……………………………………… 9

Figure II.6: Various Methods to Represent the Dependencies…………………….…. 10

Figure II.7: Polyhedral Definition and Graphical Representation of Sample Code Iteration Space……………………………………………………… 12

Figure II.8: Two Sample Codes…………………………………………………… 13

Figure II.9: Locality Enhancement of Imperfectly Nests [1]……………………..... 17

Figure II.10: Product Spaces and Mappings of Programs given at Figure II.8 [1]…… 18

Figure II.11: Mapping Representation with Unknown Coefficients…………………. 19

Figure II.12: Example Dependencies……………………………………….……… 20

Figure II.13: Example to Polyhedral Representation of Reuses…………………….. 22

Figure II.14: An Example Transformation and Mapping that cause this Transformation 22

Figure III.1: The Pseudocode for Simple GA ......…………………………………… 24

Figure III.2: Single and Double Point Recombination Operators……………………. 26

Figure III.3: Possible Places to Insert Problem Specific Knowledge and Other Operators in GA lifetime [10] ………………………………………... 28

Figure III.4: The Pseudocode for Hybrid GA [10] ....……………………….… 28

Figure IV.1: Flow Diagram of Our Optimization Framework…………………….… 32

Figure IV.2: The Steps of the Proposed Framework…………………………………. 33

Figure IV.3: An Example Original Code for Triangular Solve……………………….. 33

Figure IV.4: Example Token Array Elements for the Sample Code………………….. 34
Figure IV.5: Pseudocode for Construction of Iterations and Statements……………... 34
Figure IV.6: Values Stored at the Statement Structures for the example given at Figure IV.3………………………………………………………………….. 35
Figure IV.7: Polyhedral Representations of Dependencies…………………………… 36
Figure IV.8: Polyhedral Representations of Reuses……………………………….. 37
Figure IV.9: Affine representation of first dependence in the example………….…… 38
Figure IV.10: (a) Content of the Input File to Omega Calculator (b) Content of the Output File from Omega Calculator……………………………….…. 39
Figure IV.11: Valid and Invalid Solutions for the example given at Figure IV.3 ……. 41
Figure IV.12: An Example to the Omega Calculator Code Generation Facility, and the Mappings that Causes This Transformation…………………... 42
Figure IV.13: Solution Representation of Mappings in Their Affine Form………….. 43
Figure IV.14: Our Initial Population Generation Algorithm……………………….. 44
Figure IV.15: Example Crossover Operation with Crossover Point 3 over the First Statement Mappings…………………………………………… 45
Figure IV.16: Example Mutation Operation with Points (4,3)………………………... 46
Figure IV.17: An example offspring and its neighbor………………………………… 47
Figure IV.18: Fitness Values for the example given at Figure IV.3 . ..……………….. 48
Figure V.1: (a) Original Code and (b) Transformed Code for Triangular Solve……… 51
Figure V.2: Generated Mappings for Two Statements of Triangular Solve………….. 51
Figure V.3: Graphical Performances for Triangular Solve …………………………… 52
Figure V.4: (a) Original Code and (b) Transformed Code for Red-Black Gauss-Seidel Relaxation……………………………………………………… 53
Figure V.5: Graphical Performances for Red-Black Gauss-Seidel Relaxation………. 54
Figure V.6: Generated Mappings for Two Statements of Red-Black Gauss-Seidel….. 55
Figure V.7: (a) Original Code and (b) Transformed Code for Jacobi………………… 55
Figure V.8: Graphical Performances for Jacobi……………………………………. 56
Figure V.9: Generated Mappings for Two Statements of Jacobi…………………….. 57
Figure V.10: (a) Original Code and (b) Transformed Code for Cholesky…………… 58
Figure V.11: Graphical Performances for Cholesky Factorization…………………… 58
Figure V.12: Generated Mappings for three statements of Cholesky Factorization…. 59

Figure A.1: System of Linear Equalities with *m* Constraints and *n* Unknowns……….. 64
Figure A2: Two Examples of Applying Fourier Motzkin Elimination……………….. 66
Figure B.1: An Example to Implementation of Farkas Lemma……………………….. 68
Figure C.1: Relation Definition by Using (a) Omega Calculator, (b) Omega Library… 71
Figure C.2: Some example to the OC methods……………………………………... 72

LIST OF TABLES

PAGE NO

Table V.1: Numerical Performances for Triangular Solve………………………….. 52

Table V.2: Numerical Performances for Red-Black Gauss-Seidel Relaxation …….. 54

Table V.3: Numerical Performances for Jacobi …………………………………….. 56

Table V.4: Numerical Performances for Cholesky Factorization……………………. 58

PART I

INTRODUCTION

Developing optimization techniques for compilers become one of the common goals of compiler community for years. Although, there are various optimization methods proposed for the compilers so far, it is not practical and possible of selecting the best set of optimizations for each given code. As a broader categorization, compiler optimization techniques can be grouped as: (a) locally or globally scoped; (b) they are programming language dependent or independent; and (c) machine dependent or independent.

Optimization can be applied various scopes, from a single statement to an entire program. Generally locally scoped techniques are considered due to their simplicity and importance; since, many programs spend a large percent of their time inside loops especially image processing, video transmission, scientific programs etc.Most languages share common programming structures, such as decision blocks (if, switch, case), looping structures (for, while, do … while). Thus similar optimization techniques can be used across languages. However some language features make some kinds of optimizations impossible and/or difficult. For instance, the existence of pointers in C and C++ makes certain optimizations of array references difficult. Many optimizations are dependent to the machine targeted by the compiler. For instance, the machine cache size, memory hierarchy,

cache/memory transfer rate are important while determining the loop transformation parameters.

Processors are faster than memories according to their speeds. To overcome memory speed problem, manufacturers designed memory systems in a hierarchy. But this hierarchical design is effective when program exhibit data locality. Cache misses are the main negative effect on program performance. The major reason of cache misses is the nested loops. In scientific applications, there are loop nests, which cause the most time and memory consuming. Especially the application includes like image processing, computational fluid dynamics, geophysical data analysis has large amount of loop nests. So loop transformations play an important role in the compiler optimization techniques for these types of applications.

Iterative compilation techniques [7, 8] are proposed in the literature, which target to choose the type of loop transformation to apply and to decide the best parameter set for the selected transformations. The parameter space is quite large and choosing the best parameter combination like tile size, unroll factor etc, make the problem more difficult. By applying the polyhedral equations and representing iteration spaces at the recent studies [1, 2], the quality of solutions become better. Additionally, general loop transformations can be represented with matrix form. The polyhedral representation gives us a change to create unified framework for implementation of generic program transformations.

In this study we proposed a compiler optimization framework for improving data locality of the programs with the objective of minimizing the execution time of the programs, where the representation of loop nests and statements in our framework is based on the polyhedral model [1, 2]. Our compiler optimization framework is not machine dependent since the generated code is not run on a machine and execution time of the program is not used as our fitness value of the solution. Our target is finding the optimal code for many machine architecture types by improving the data locality at source level. Additionally, it is a generic one that it is not tried to find good loop transformation parameters like tile size, unroll factors described at section II.2. We considered all types of loop transformations. The framework is not language dependent, since the parse part of the source code can be

changed to any other languages. Another important difference of our framework is that we process all statements and loop nests in a procedure not only one loop nest.

Our proposed framework is based on a hybrid genetic algorithm. Genetic Algorithms (GAs) [10, 11] are stochastic search methods that use the natural selection principle. The best solution is found by generating various solutions from the current generation by applying the recombination and mutation operators. These operators provide the survival of the fittest principle. GAs have been widely used in the application area of science, engineering and business [12]. GAs are usually considered with the problem specific knowledge [10, 28] since simple GAs are generally weak for solving the complex problems [24]. GAs can be used with conjunction of problem specific knowledge in various ways [10]. Our algorithm uses a problem specific heuristic to set the initial population; and local search technique is applied to improve generated offsprings.

The experimental evaluation based on real application codes reveals that our hybrid genetic algorithm generates good results as much as compiler optimizations and at some cases it outperforms the compiler optimizations. When we do not apply the compiler optimization to our transformed code, this choice always gives the minimum execution time.

I.1 OUTLINE OF THE THESIS

The rest of this thesis is structured as follows. Chapter 2 discusses the main issues about the compiler optimization problem by applying loop transformations. In this chapter, firstly compiler optimization techniques and the related terminology are given in details. After describing known loop transformations, two main approaches for finding the best loop transformation sequence are presented.

Chapter 3 gives general information about hybrid genetic algorithms. This chapter firstly describes the principle of genetic algorithm and its usage area then hybrid genetic algorithms and its characteristics are given in the subsection.

Chapter 4 presents our hybrid algorithm in details. String representation, initial population, local search method and our problem specific heuristics are given in subsections.

Chapter 5 gives the result of our experimental evaluation. The source code and the generated codes are compiled with “SUN” and “GNU” compilers. Also two types of compilation done for both of them: by opening compiler optimization flag and by closing compiler optimization flags. The results of our implementation clearly show that the proposed algorithm is successful in improving data locality in nested loops.

Finally Chapter 6 summarizes the conclusions of our study and gives brief information about data transformation techniques that could be addressed as a part of a future work.

PART II

COMPILER OPTIMIZATION TECHNIQUES

In this part, we briefly explain the compiler optimization related terminology. The first section gives a detailed explanation of cache terminology. At the second section, we present the details of loop transformation techniques that are widely used in the compiler optimization area. The third section describes the concepts on preserving the semantic of program in the compiler optimization problem. The fourth section presents the techniques for measuring performance of programs. Then we present the polyhedral representation approach to the problem. Finally, related works on compiler optimization are presented at the last part.

II.1. CACHE RELATED TERMINOLOGY

There is a growing gap between the cycle time of a processor and the access time of the main memory, which is due to the recent advances and trends in computer architectures. To overcome this bottleneck, a ***cache*** is employed in processors commodity. The cache is a fast memory, which is located close to the CPU. Generally speaking, data can be loaded from/stored to the cache in 1 - 2 clock cycle. Data transformation between cache and memory can requires 20 - 100 clock cycles or more [31]. So if a data in the cache is reused several times, high efficiencies are possible.

Data locality is the sequential data accesses between the neighbor address spaces. ***Miss count*** is the number of data transformations between the memory and cache, when the data is not found in the cache. The data of very small applications

will fit entirely in the cache, and be very efficient. Larger applications having the property of data locality will have a high cache hit and low miss count, will also be quite efficient. If the cache hit decreases and miss count increases, then program performance degrades rapidly. A high performance computer typically has a larger cache than a typical PC, which increases the data locality and cache hits, and also decreases miss counts. Since the memory system may limit the performance of programs running on modern architectures instead of the speed of the processors, efficient use of memory systems is one of the key issues to improve the performance of the programs.

There exist two different types of locality: ***temporal locality (locality in time)*** says if an item is referenced, it will tend to be referenced again as soon, and ***spatial locality (locality in space)*** says if an item is referenced, items whose addresses are close by tend to be referenced soon.

A compiler can restructure applications to uncover and enhance locality in two different methods: it can change the order of memory accesses or, it can change the layout of memory. The most common way is the first one. Applying loop transformation changes the order of memory accesses. Also rewriting loops make better use of the memory system by increasing the number of instructions executed, which can degrade the run-time performance of some programs. Array references within the loops are used in the loop transformation optimizations. These optimizations can improve the performance of the memory system and usually apply to multiple nested loops [1, 2]. The second way, changing the array layout in memory, is done by data transformations like array padding [15, 16]. The main idea is to transform the array layouts in memory that provides two loop iterations executed sequentially access the data in the same memory location as much as possible. The second method is out of scope of this thesis.

```
    for i = 1, N
        for j = i+1, N
S1:         A(j) = A(j) / A(i)
        end
    end
```

```
        for i = 1, N
S1:         A(i) = sqrt(A(i))
            for j = i+1, N
S2:             A(j) = A(j) / A(i)
            end
        end
```

Figure II.1: (a) A Perfectly Nested Loop, and (b) An Imperfectly Nested Loop

Loop transformations improve the performance according to the memory system; by decreasing the cache miss counts. Loop nests are categorized as perfectly or imperfectly nested loops in the literature. In perfectly nested loops, all statements are contained within the inner most loop of a loop nest. Unfortunately, most loops in real programs are an imperfectly nested loop that is the assignment statements can be any where in the loop nest. Figure II.1 shows an example to perfectly and imperfectly nested loops.

II.2. LOOP TRANSFORMATION TECHNIQUES

Loop transformations techniques try to change order of memory accesses to increase data locality and program performance. There are several methods for implementing loop restructuring like loop fusion, loop fission, loop tiling, loop unrolling etc [3, 18]. In this section we present the details of the four selected methods.

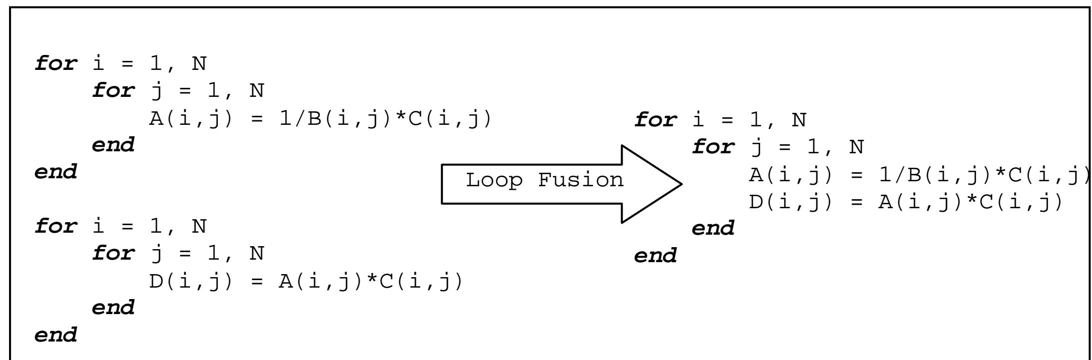


Figure II.2: Example Code for Loop Fusion

Loop Fusion: Loop fusion consists of combining adjacent or closely located loops into a single loop [29]. Some programs have separate sections of code that access the same arrays and perform different computations on common data. By fusing multiple loops into single loop allow the data in cache used repeatedly before transferring the memory. This method reduces cache misses generally by exploiting the temporal locality rather than spatial locality. Figure II.2 shows an example loop fusion. In this example, without loop fusion, accessing array A and C increases the number of misses by twice.

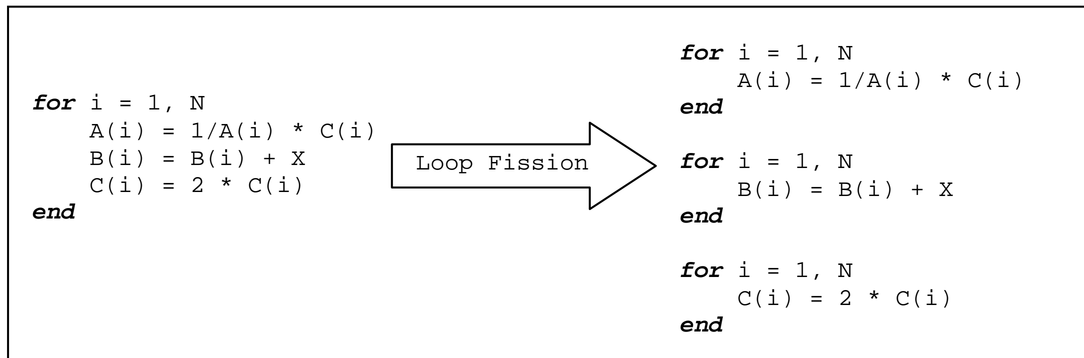


Figure II.3: Example Code for Loop Fission

Loop Fission: Loop fission is the opposite of loop fusion. A loop is split into two or more loops [29]. This optimization is appropriate if the number of computations in a loop becomes excessive, leading to register spills that degrade performance. So loop fission can improve memory locality as better use of instruction cache or memory cache. Also loop fission may facilitate other transformations. Sometimes the compiler applies loop fission to split a loop apart, and then performs loop fusion to recombine the loop in a different way to increase performance. Figure II.3 shows an example to loop fission. In this example, loop fission gets benefit from memory locality since the accesses to the arrays A, B and C are doing in different loop iterations.

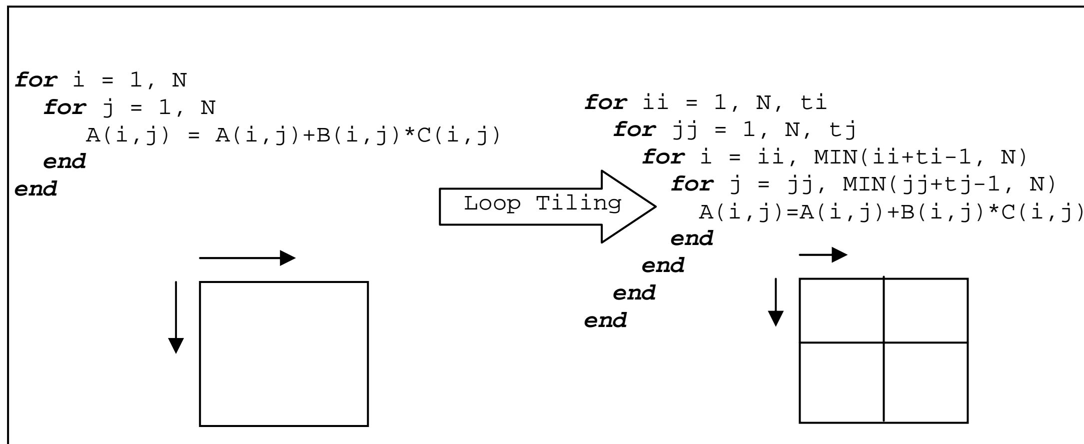


Figure II.4: Example Code for Loop Tiling

Loop Tiling: Tiling divides the iteration spaces into tiles and loop nests iterate over them [29]. By iteration over the sub iteration spaces, the use of cache improves. It improves the temporal data locality and reduces the number of data cache misses. Tiling enhances the data locality by reducing the number of iterations between uses

of the same data. At Figure II.4, the i and j loops are tiled by tile factor ti and tj, respectively. This transformation can be used in cases where the amount of data touched within this loop nest is bigger than the cache size. Wolfe describes the legality issues of loop tilling and gives practical example of this transformation [18]. Choosing the suitable tile size that can improve the performance become very important when applying loop-tiling transformation.

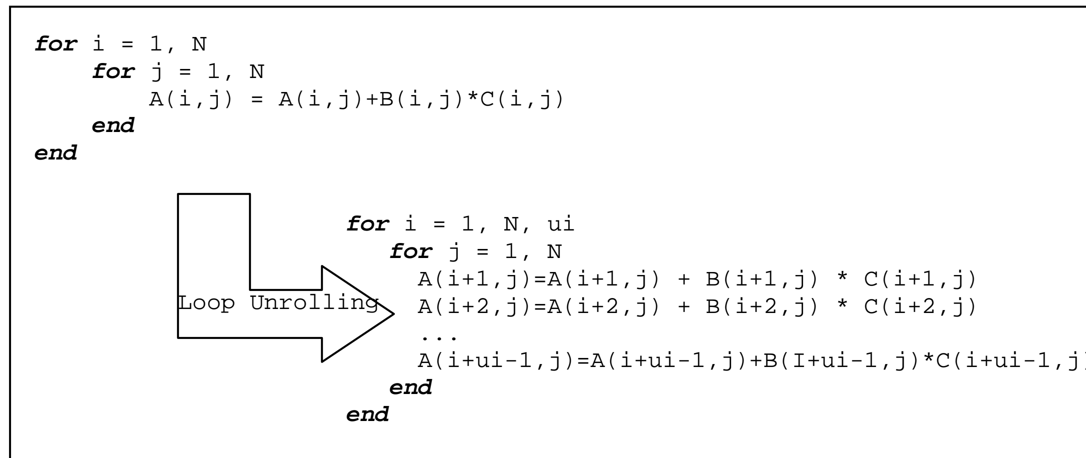


Figure II.5: Example Code for Loop Unrolling

Loop Unrolling: In some programs loops have such a small body that most of the time is spent to increment the loop-counter variables and to test the loop-exit condition. We can make these loops more efficient by unrolling them, putting two or more copies of the loop body in a row. As the body of the loop becomes larger, the compiler can schedule the instructions more efficiently. For example, instead of using array element a(i) in one statement, the loop has three statements using elements a(i), a(i+1), a(i+2) and increments i by 3. This divides the loop overhead by the amount of unrolling, but more importantly gives the compiler more statements in the basic block to optimize. This can improve instruction scheduling and reduce memory access time. For example, at Figure II.5 the loop i unrolled by unroll factor of ui. The unroll factor is the number of times that the loop body is copied [3].

II.3. DATA DEPENDENCE CONCEPT

Applying transforming must preserve the semantics of the program. This can be done by finding the dependencies in the program and applying the transformation by considering these dependencies. There are three main dependence types given in the literature [29]**.** Note that X and Y variables in the following examples and dependence occurs at X.

- *Flow Dependence*: It occurs when a variable is assigned in a preceding statement and used in a successor statement.

  S1: X = 1

  S2: Y = X

- *Anti Dependence*: It occurs when a variable is used in a preceding statement and assigned in a successor statement.

  S1: Y = X

  S2: X = 1

- *Output Dependence:* It occurs when a variable is assigned in both preceding and successor statements.

  S1: X = 1

  S2: X = 2

Example Code:	Graph :	Symbolic:	Set:
S1: A = 10	S1	S1 δf S3	OUT(S1) $\cap$ IN(S3) $\neq \emptyset$
S2: B = C + 10	S2	S1 δf S4	OUT(S1) $\cap$ IN(S4) $\neq \emptyset$
S3: C = A	S3	S2 δa S3	IN(S2) $\cap$ OUT(S3) $\neq \emptyset$
S4: D = A + C	S4		

Figure II.6: Various Methods to Represent the Dependencies

The three well-known methods for representing these dependencies are graphical, symbolic and set representations [29]. Graphical representation is a good way since it allows us to see the dependences easily. In graphical representation, a plain arrow between statements is used to represent flow dependence. An arrow with a crossing line is used to represent anti dependence. At symbolic

representation, the character sigma (δ) and the initial character of the dependences are used to represent dependencies. In set representation two sets IN() and OUT() are declared. IN(S_i) is the set of variables that is read by the statement i and OUT(S_i) is the set of variables that is modified by the statement i. Figure II.6 gives the three representations of the given program fragment. In this thesis we uses the set representation of dependences.

II.4. MEASURING THE LOCALITY ENHANCEMENT

There are various techniques or metrics that can be used to measure the "goodness" of an execution order [1]. It is clear that the most exact measurement of the enhancement is to execute the transformed code. Large solution space makes it infeasible to execute every possible enhancement. Moreover, it makes the algorithm machine dependent; and different machines find different best solutions according to their architectures.

Another type of measurement is to count number of cache misses. Since much more time is spending while program running when cache miss occurs and tries to get data from memory, the count of cache misses can show the performance. Special hardware solutions exist to count the cache misses but it is not also feasible, since again the execution of candidate transformations must be executed to count cache misses. There are also some programs, which can model the cache misses accurately [4]. To calculate cache misses it requires a lot of parameters, including cache size, cache associativity, cache policy; but these parameters make algorithm machine architecture dependent.

There is another algorithm, which calculates the count of cache lines that a loop nest accesses. But it is limited, since it assumes all loops are perfectly nested [9]. They count the number of cache lines accessed by a loop nest with the help of a formulation. They grouped the array accesses, which can exhibit temporal or spatial locality and count the probable cache misses according to the loop nests that surrounding these reference groups. In this approach, they assume there is no cache misses in the inner most loops.

Another measurement strategy is based on polyhedral representation model (see section II.5) [1, 2]. Statement iterations are modeled on iteration spaces by

using matrixes. Data reuses are defined as accessing the same memory location from different statement instances. So the data reuse distances can be shown by difference vectors of statements. As a result, this approach states that the data reuse distances can be used for our cost function and aim is to reduce reuse distances by making reuse difference vectors' entries zero. If the data reuse difference vectors are considered as the measurement of enhancements, it does not depend to the machine architectures. In our study, we consider reuse distances as the metric of measuring the locality enhancement.

II.5. POLYHEDRAL MODEL

Previous approaches to iterative compilation try to find best loop transformation sequences by searching in a small set of loop transformations [6, 7, 8]. Current approaches are generally uses more generic methods to find combination of transformations not only one transformation [1, 2]. The polyhedral modeling of programs are used to produce more generic framework for optimization. In a polyhedral model, linear algebraic representations are used to model a program. This model is very powerful since it benefits from mathematical theory and geometric representation. Generally, a polyhedron stands for a convex set of points in a lattice. A set of points in a Z vector space bounded by affine inequalities defined as follows [21]:

$$\text{Iteration Space} = \{ x \mid x \in Z , Ax \geq b\}$$

where x is the iteration vector, A is a constant matrix and b is a constant vector.

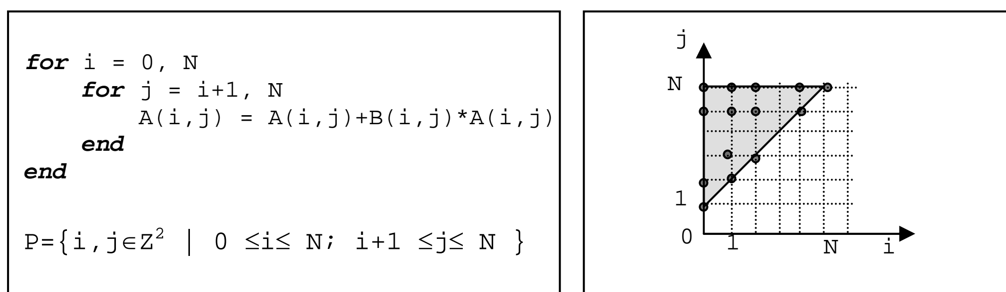


Figure II.7: Polyhedral Definition and Graphical Representation of Sample Code Iteration Space.

The polyhedral model allows manipulating loop nests with their bounds as affine functions of integer-valued parameters [22]. While loop indices are incremented by a constant integer value, the values of indices form an n-vector, which belong to a subset of Z^n: an integer lattice L. The values are taken from the index vector $I \in Z^n$, where n is the number of nested loops. The loops nests are transformed by changing the Z polyhedron representation [23]. Figure II.7 shows a sample code iterations modeled by parameterized Z^2 polyhedron, where N is a constant.

II.5.1. Program Execution Model by Polyhedral Method

Statements iteration spaces in nested loops are defined as a polyhedron by a set of linear inequalities. Whole program iteration space can be obtained by combining individual statement iteration spaces [1, 2]. The statement execution points in the program iteration space can also be obtained by using mapping functions for all statements. These mappings are affine functions which caries the statements from their iteration spaces to program iteration spaces. Then the optimization aim is finding the good mappings that preserve the semantics of the program.

A program consists of statement and statements are named as $S_1, S_2, ..., S_n$ in syntactic order. $S_k(i_k)$ is referred as the dynamic instance of the statement k, for value of index variables i_k. So we can show dynamic instance of executing the programs given at Figure II.8 as the followings:

Program 1: $\{ S_1(1), S_1(2), S_1(3), ..., S_1(N), S_2(1), S_2(2), S_2(3), ..., S_2(N)\}$

Program 2: $\{ S_1(1), S_2(1), S_1(2), S_2(2), S_1(3), S_2(3), \ldots, S_1(N), S_2(N)\}$

Program 1:	Program 2:

<pre>   for i = 1, N
S1   A(i) = B(i)
   end
 	
   for i = 1, N
S2   A(i) = A(i) + B(i) * C(i)
   end</pre>
	<pre> for i = 1, N
	S1 A(i) = B(i)
	S2 A(i) = A(i) + B(i) * C(i)
	   end</pre>

Figure II.8: Two sample Codes

The iteration spaces for the sample programs at Figure II.8 are:

Program 1: Iteration space of $S_1 = \{ i_1 \mid i_1 \in Z , 1 \le i_1 \le N \}$

Iteration space of $S_2 = \{ i_2 \mid i_2 \in Z , 1 \le i_2 \le N \}$

Program 2: Iteration space of $S_1 = \{ i_1 \mid i_1 \in Z , 1 \le i_1 \le N \}$

Iteration space of $S_2 = \{ i_1 \mid i_1 \in Z , 1 \le i_1 \le N \}$

As you can see there exist two different iteration space, i_1 and i_2, for the first program, but only one iteration space, i_1, for the second program. At the first one, since i_1 and i_2 iteration spaces proceed at different times, when i_1=1 there is only one statement mapped to this point in iteration space which is $S_1(1)$. At the second one S_1 and S_2 statements have the same iteration spaces and the order of statements became important at execution. In another words when $i_1 = 1$, there are two statements mapped to point i_1=1, so the execution order of them is determined according to the statements order and $S_1(1)$ executes before S_2 (1).

II.6. RELATED WORKS

In this section we give details of two approaches on loop transformations. The first subsection presents the iterative compiler optimization technique that are intended to find good loop and data transformation parameters such as till size, unroll factor [6,7,8]. The second section presents recent studies for compiler optimization that targets to find good sequence of loop transformations in addition to the parameters by using polyhedral model [1,2]. By representing the transformations as matrixes, they catch more flexible way of optimization. Their aims are only generating good sequence of transformations that enhance the data locality and do not change the program semantic.

II.6.1. Iterative Compilation

Consequently, software optimization methods based on program transformations are used to improve data locality and reduce the number of cache misses. Some major program transformations can be applied manually for small and simple programs. However, it requires a good and detailed knowledge of the underlying hardware from a programmer and is a tedious and time-consuming

process. Moreover, any small changes in the software or hardware parameters may invalidate the whole optimization process. Therefore, automatic optimization approaches are desirable for optimizing portable codes for particular architectures.

Traditional automatic optimization approaches are based on comprehensive static program analysis which analyses program data locality and to predict the number of cache misses, taking into consideration software parameters and hardware models. Static approaches fail to select the best optimization and they fail in cases where information is not available at compile time.

Dynamic methods are intended to solve these problems by obtaining various runtime parameters during program execution and then by re-optimizing this program using these parameters. Smith describes a tool called Pixie in [20] that collects the runtime information about the program basic blocks. This tool rewrites the executable file and inserts additional instructions to count the number of executions of each basic block. The dynamic methods fail to tackle the problem of the growing performance gap between processor and memory.

Iterative compilation optimization approach overcomes these problems [7, 8, 30]. This approach creates variants of the program with different transformation parameters. All variants of the program are executed and the one with the lowest execution time is picked as the best version. The major advantage of this approach is that it does not need the detailed knowledge of the program and the underlying hardware. The major disadvantage is the excessive compilation time of iterative methods.

Iterative compilation studies search the large transformation parameter space so the search algorithms make difference between them. Since the goal function, minimization of execution time cannot be estimated at compile time, the optimization is difficult. These techniques generally run the program at compile time and get the feedback until the good transformation parameters are found. In the embedded applications this can be acceptable since these applications compiles one time and runs many times.

II.6.2. Generic Polyhedral Methods

In this section, we give details about the recent study [1] on program optimization by using polyhedral methods.

II.6.2.1. Locality Enhancement for Imperfectly nested Loops

In this section we present the details of Nawaaz Ahmed`s work [1,17], which extends the techniques proposed for perfectly nested loops for locality enhancement of imperfectly nested-loops. Polyhedral algebra is used to represent program execution. The loop nests iterations are represented as a point in an integer lattice as discussed at section II.5. Then good loop transformation sequences are represented as nonsingular matrixes, which transform one lattice to another. To overcome imperfectly nested loop problem the loops are transformed the perfectly ones by introducing the product space concept. The three main problems that they come across are [1]:

1. Determining representation of imperfectly nested loop and loop transformations
2. Determining legality of loop transformations
3. Describing and finding transformations that enhance locality

By transforming all statement iteration spaces to the product spaces they solve the imperfectly nested loops representation problem. By applying transformation from product space to another product space they solve the loop transformation representation problem. The second problem is to check the legality of the generated transformation, which is done by considering an implementation of Farkas' Lemma (which is given in detail at Appendix B.). At the end, the measurement of the locality enhancement is made by maximization of reuse distances in the program, which is given in detail at section II.4.

Nawaaz Ahmed`s strategy for solving the above mentioned problems is represented in Figure II.9. At the beginning, the statement iterations spaces are generated like at step 2. Then they are combined to produce product space for the loop nests at step 3 by using affine embedding functions F's. The representation of product space at step 3 is the corresponding to the original source code. To implement transformations like loop tiling, loop unrolling, etc. the transformation matrix T's are used. T is a matrix, which is used to convert one product space to another. The dependences must be considered and the transformation matrix must checked by using dependences. After finding legal transformation matrix, the new product space can be produced by multiplication of transformation matrix with the old product space at step 4. The aim of this study is to find the best transformation

matrix, which enhances the data locality. Finally, the best solution, product space is used to regenerate transformed code in step 5. At this point, omega calculator's code generator functionality is used. Since our study is based on the technique proposed by Nawaaz Ahmed, a more detailed explanation of the steps for locality enhancement are given in the following subsections, which are based on explanations given in his thesis [1].

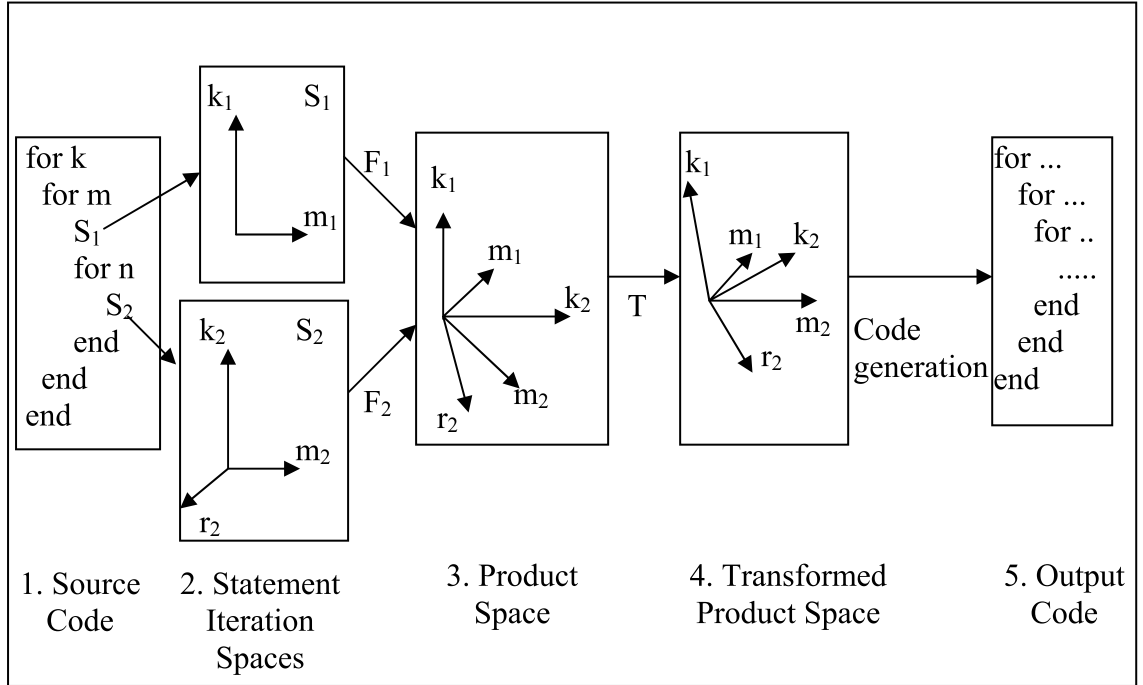


Figure II.9: Locality Enhancement of Imperfectly Nests [1]

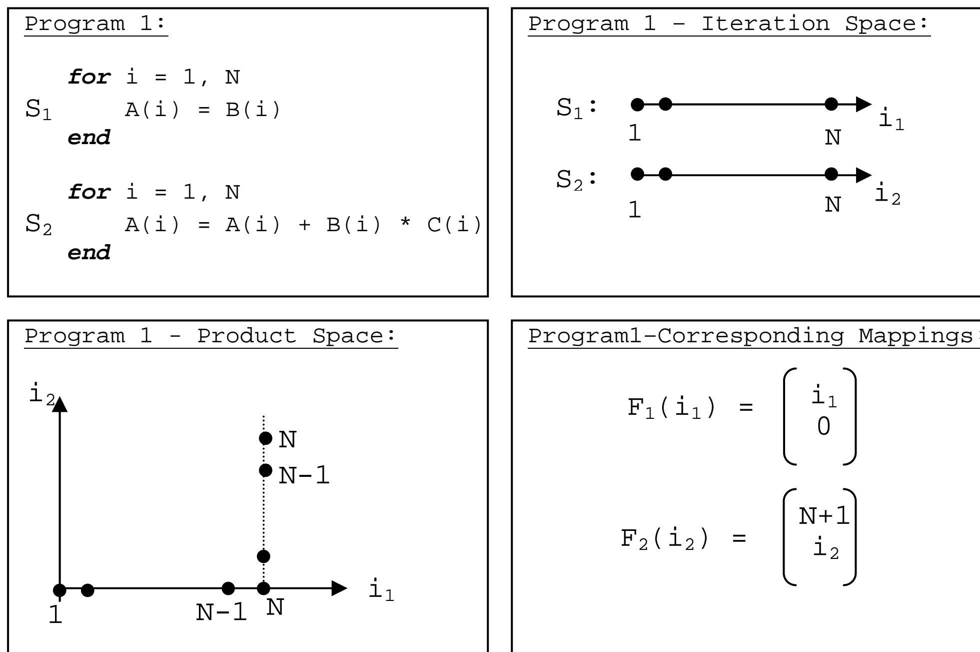
1.a. Polyhedral Representation of Iteration Spaces and Mappings

He used the program execution model defined at section II.5.1. He extends this definition to consider imperfectly nested loops. So he defined the program execution model as follow [1]:

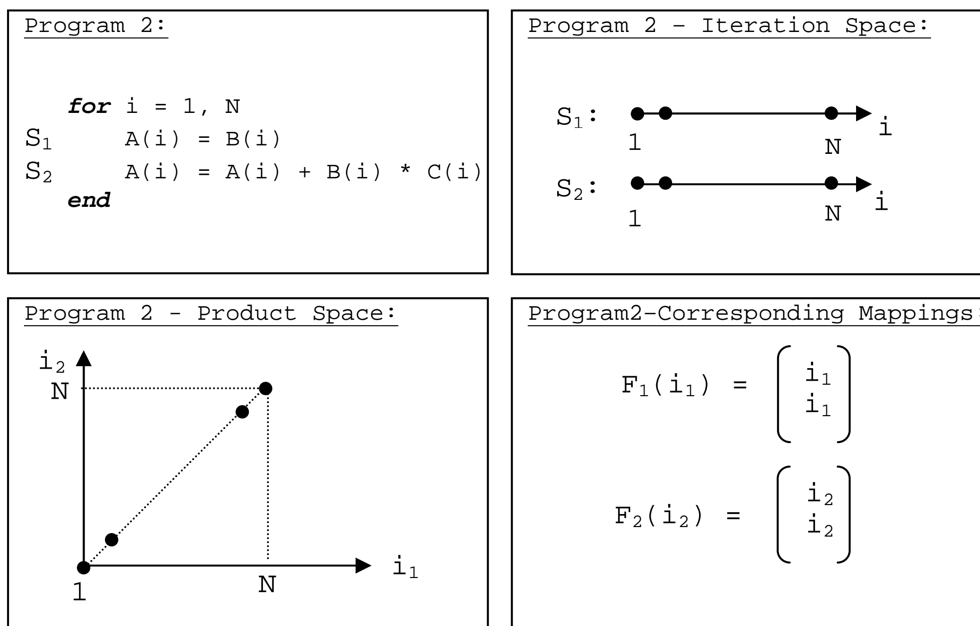
- P is product space which is formed by Cartesian product of iteration spaces
- All dynamic instances $S_k(i_k)$ can be embedded to product space by embedding functions F_k, where each F_k must be one-to-one function.

If the all points in P are traversed in lexicographic order, and all statement instances mapped to a point are executed in original order when that point is visited, the program execution order is reproduced. It can be easily seen that only P and F

pairs can model the program execution. At Figure II.10 the product spaces and mappings for the example codes at Figure II.8 are shown.



(a) Program 1 iteration space, product space and mappings



(b) Program 2 iteration space, product space and mappings

Figure II.10: Product Spaces and Mappings of Two Sample Programs given at Figure II.8 [1]

The number of dimension of iteration spaces depends directly to the number of loop nests that surrounding the statement. For instance, at Figure II.9 the statement 1 iteration space has two dimensions $\{k_1, m_1\}$ and statement 2 has three dimensions $\{k_2, m_2$ and $r_2\}$. The product space has five dimensions, which are $\{k_1, m_1, k_2, m_2$ and $r_2\}$. The sum of iteration space dimensions gives us the product space dimensions. Also affine embedding functions, F_1, F_2, ... F_n, cannot drive more dimensions than the product space. They only map a point in iteration space to a point in product space.

$$F_1\left(\begin{pmatrix} i_1 \end{pmatrix}\right) = \begin{pmatrix} G_{1i1}\ i_1 + G_{1N}\ N + g_1 \\ G_{2i1}\ i_1 + G_{2N}\ N + g_2 \end{pmatrix} \qquad F_2\left(\begin{pmatrix} i_2 \end{pmatrix}\right) = \begin{pmatrix} G_{1i2}\ i_2 + G_{1N}\ N + g_1 \\ G_{2i2}\ i_2 + G_{2N}\ N + g_2 \end{pmatrix}$$

Figure II.11: Mapping Representation with Unknown Coefficients

General representation of mappings from one-dimensional to two-dimensional shown at Figure II.11. G's are the unknown coefficients of mappings. The goal is to find most suitable G's to make data locality better. For example at Figure 11 (b), $G_{1i1} = 1$, $G_{1N} = 0$ and $g_1 = 0$ for the first dimension of first statement mapping.

1.b. Polyhedral Representation of the Dependencies

In the nested loops the dependence analysis are more complicated. Using explained dependence definitions at section II.3 we can define dependence relations for loop nests as follows: dependence exists from instance i_s of the S_s to instance i_d of the statement S_d if the following conditions are satisfied:

1. S_s and Sd has a same memory location, and at least one of them writes to that location.
2. i_s and i_d belongs to the iteration domains S_s and S_d, respectively.
3. S_s is executed before S_d.

By using these definitions, a polyhedral representation of each dependence relation between two-statements can be described with affine inequalities. So the dependence polyhedral has the following components [1]:

1. Same memory location: Array references can be expressed by using affine expressions of loop variables. Then a dependence exists if $A_s*i_s+a_s = A_d*i_d+a_d$.
2. Loop Bounds: Iteration space bounds are affine expression of index variables. So they can be expressed as $B_s*i_s+b_s \geq 0$ and $B_d*i_d+b_d \geq 0$ for matrices B_s, B_d and vectors b_s, b_d.
3. Precedence order: The instance is of Ss executed before instance id of Sd in program execution order can be expressed as $X_s*i_s - X_d*i_d + x \geq 0$.

As a result, if these inequalities are combined, the dependence matrix is produced. Each matrix inequality will be called a dependence class and will be denoted by D with appropriate subscript. So the set of linear inequalities which capture the constraint between source and destination shown below:

$$D \begin{pmatrix} i_s \\ i_d \end{pmatrix} + d = \begin{pmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{pmatrix} \begin{pmatrix} i_s \\ i_d \end{pmatrix} + \begin{pmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{pmatrix} \geq 0 \qquad (\text{II.1})$$

Shortly the dependence classes can be represented as follows:

$$D\text{:} \quad D \begin{pmatrix} i_s \\ i_d \end{pmatrix} + d >= 0$$

Code for Triangular Solve:	Dependence 1:	Dependence 2:

`for i = 1, M`
`  for j = 1, N`
`    for k = 1, j-1`
`S1:   A(j,i)=A(j,i)-B(j,k)*A(k,i)`
`    end`
`S2:  A(j,i) = A(j,i) / B(j,j)`
`  end`
`end`
	S1 loop bounds:
	1 ≤ i1 ≤ M
	1 ≤ j1 ≤ N
	1 ≤ k1 ≤ j1-1
	
	S2 loop bounds:
	1 ≤ i2 ≤ M
	1 ≤ j2 ≤ N
	
	Array A indices:
	j1 = j2
	i1 = i2
		S1 loop bounds:
		1 ≤ i1 ≤ M
		1 ≤ j1 ≤ N
		1 ≤ k1 ≤ j1-1
		
		S2 loop bounds:
		1 ≤ i2 ≤ M
		1 ≤ j2 ≤ N
		
		Array A indices:
		k1 = j2
		i1 = i2

Figure II.12: Example Dependencies

An example program is given is shown at Figure II.12, which has two dependencies. The first dependence exists since statement S_1 writes to location A(j,i) which is then read by statement S_2. Also there is a dependence between statement S_2 and S_1, since S_2 writes to location A(j,i) which is then read by A(k,i).

Dependence classes are used to check legality. If (P, $\{F_1, F_2, \ldots F_n\}$) is representing the execution order of a program then we can say that the embedding functions are legal if the difference vector of the embedding functions in dependence classes, $F_d(i_d) - F_s(i_s)$, is greater than or equal to zero. In other words, the point that i_s mapped must be earlier than the point that i_d mapped.

1.c. Measuring the Enhancements on Imperfectly Nested Loops

In the related work, Ahmed et al. used the polyhedral model representation of reuses between the array references in order to measure the locality enhancements. The polyhedral representation and using linear algebra makes this method easier than other techniques proposed [1].

As dependence classes the reuse classes can be represented in polyhedral from by using the inequalities, which comes from the definition of reuses. A reuse exists from instance i_s of the statement S_s to the instance i_d of the statement S_d if the following conditions satisfies [1]:

1. Same memory location: Both statement instances have a reference to the same memory location. If we consider memory locations as array references then a reuse exists if $A_s*i_s+a_s = A_d*i_d+a_d$.
2. Loop Bounds: Both statement instances lies on the corresponding iteration spaces. We can represent the iteration spaces bounds as $B_s*i_s+b_s \geq 0$ and $B_d*i_d+b_d \geq 0$ for matrices B_s, B_d and vectors b_s, b_d.
3. Precedence order: The instance is of S_s executed before instance id of S_d in program execution order can be expressed as $X_s*i_s - X_d*i_d + x \geq 0$.

So the set of linear inequalities which capture the constraint between source of the reuse and destination of the reuse shown below:

$$R \begin{pmatrix} i_s \\ i_d \end{pmatrix} + r = \begin{pmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{pmatrix} \begin{pmatrix} i_s \\ i_d \end{pmatrix} + \begin{pmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{pmatrix} >= 0 \qquad (\text{II.3})$$

This definition of reuses is for the temporal reuses since first condition expresses the same memory locations. The spatial reuses, accessing the nearby memory locations, can be considered by changing the condition as $A_s*i_s+a_s+c = A_d*i_d+a_d$, where c is the number of array element that fit into a single cache line. We show an example to the reuse representation at Figure II.13. There exists reuse between statement S_1 and S_2 since they touch the same array reference A.

Example:	Reuse between S1 and S2:

`for i = 1, N`
`S1    A(i) = B(i)`
`end`
`for i = 1, N`
`S2    A(i) = A(i)+B(i)*C(i)`
`end`
	S_1 loop bounds:
	$1 \le i_1 \le N$
	S_2 loop bounds:
	$1 \le i_2 \le N$
	Array A indices:
	$i_1 = i_2$ for temporal reuse, or
	$i_1 = i_2+4$ for spatial reuse
	
	$R1 = \{(i_1,i_2):1 \le i_1,i_2 \le N,\ i_1=i_2\}$ or
	$R2 = \{(i_1,i_2):1 \le i_1,i_2 \le N,\ 0 \le i_2-i_1 \le 4\}$

Figure II.13: Example to Polyhedral Representation of Reuses.

Original Code:	Transformed Code (Permuted):	Mapping:

`for i = 1, M`
`  for k = 1, N`
`S1:  A(k) = A(k) + k`
`  end`
`end`
	`for k = 1, N`
	`  for i = 1, M`
	`S1:  A(k) = A(k) + k`
	`  end`
	`end`
		$F_1\left(\begin{pmatrix} k \\ i \end{pmatrix}\right) = \begin{pmatrix} i \\ k \end{pmatrix}$

Figure II.14: An Example Transformation and Mapping that cause this Transformation.

Let's look at the example of loop permutation that reduces the reuse distance, at Figure II.14. Array reference A[j] is accessed each time for the iterations of i loop. The reuse distance between the accesses to A[j] is N. We can decrease the reuse distance by permuting i loop. Now the code that loop permutation is applied has the ruse distance 0 because of the A[j] reference. At the near of transformed code, you can see the mappings which cause this transformation.

PART III

HYBRID GENETIC ALGORITHMS

Evolutionary algorithms (EAs) are a broader class of stochastic search methods that are based on principles of natural genetics. They have been applied successfully in many problems on different areas, such as search, optimization, and machine learning problems [10, 12, 13]. There are many variations of evolutionary algorithms presented in the literature [10], such as evolution strategies (ES), evolutionary programming (EP), genetic algorithms (GA), and genetic programming (GP); The EA term provides a common term for all of its variants.

In this chapter, we present an overview on genetic algorithm, which is followed by the details of the hybrid evolutionary algorithms called Memetic Algorithms (MA).

III.1. GENETIC ALGORITHMS

When the applications of GA are examined, it can be shown that GA is particularly suited to multidimensional global search problems where the search space potentially contains multiple local minima. The basic GA does not require extensive knowledge of the search space, such as solution bounds or functional derivatives. There are two distinct classes of GA:

- Those, which have no concept of a generation, known as *steady-state* GAs
- Those, which do follow the concept of generations, known as *generational*, or *generation-based*, GAs.

In a steady-state genetic algorithm one member of the population is changed at a time. In crossover two members of the population are chosen, a single child created which replaces a member of the population. Any selection method can be used to select the individual for mutation or the parents. There are a number of different replacement strategies, replace worst (often gives too rapid convergence), replace a randomly chosen member or select replacement using the negative fitness.

On the other hand, in generational GAs the entire solution is replaced every generation by the offspring population. Therefore each individual can be alive in each generation in case of no elitism considered. The steady-state GA, therefore, appears twice as fast although it can lose out in the long term because it does not explore the landscape as well as the generational GA. Since their performance depends the particular problem, there is no common acceptance of one of them in the GA community.

1. ***initialize*** population
2. ***evaluate*** each candidate solution
3. ***while*** (termination criteria is not satisfied)
4. ***select*** two solution as parents
5. ***recombine*** parents
6. ***mutate*** offspring
7. ***evaluate*** new candidate solution
8. ***select*** solutions for the next generation
9. ***end while***

Figure III.1: The Pseudocode for Simple GA.

Figure III.1 shows pseudocode of a simple GA implementation [10]. The algorithm begins with a set of solutions called initial population. The initial population is generated by using some heuristics or randomly. Solutions represented by chromosomes. Solutions from one population are taken and used to form a new population. GA selection operators perform the equivalent role to natural selection. The overall effect is to bias the gene set in following generations to those genes, which belong to the most fit individuals in the current generation.

As mentioned above, the three main operators of GAs are selection, crossover and mutation operators. Solutions are usually selected according to their fitness to form new solutions and there are various selection operators described in the literature [10] such as: roulette wheel selection, tournament selection, rank

selection and others. Roulette wheel selection is a way of choosing members from the population of chromosomes in a way that is proportional to their fitness. Parents are selected according to their fitness. The better the fitness of the chromosome, the greater the chance it will be selected; however it is not guaranteed that the fittest member goes to the next generation. In tournament selection *n* individuals are selected at random and the fittest is selected. The most common type of tournament selection is binary tournament selection, where just two individuals are selected. The roulette method of selection will have problems when the fitnesses differ greatly. For example, if the best chromosome fitness is 75% of the entire roulette wheel then the other chromosomes will have a very small chance of being selected. In the rank selection method solutions in the population gets its rank and then every solution receives fitness from this ranking. The worst will have fitness 1, second worst 2 etc. and the best will have fitness N (number of chromosomes in population).

The algorithm tries to generate new improved solutions by implementing the *recombination* (*crossover*) operator. This is the key procedure of GA that tries to generate better solutions by combining the features of good ones. The most commonly used form is chromosome mixing, where intact chromosomes are randomly swapped, which are highly advantageous in some applications. The simplest form of crossover is one point crossover in which only one crossover point is selected. More complex crossover operator is two-point crossover in which two crossover points are randomly selected. For many real applications, problem-specific solution representations and crossover operators have been developed. This flexibility is one of the attractions of GAs. It is very easy to introduce heuristic operators, which can substantially improve algorithm performance.

Mutation operator prevents irreversible loss of genetic information and hence provides diversity within the population. For instance, if every solution in the population has 0 as the value of a particular bit, then no amount of crossover will produce a solution with a 1 there instead. Again as in the crossover operator the mutation can be done as single and double point mutation. Figure III.2 shows two different applications of crossover and mutation operations, which are called single and double point.

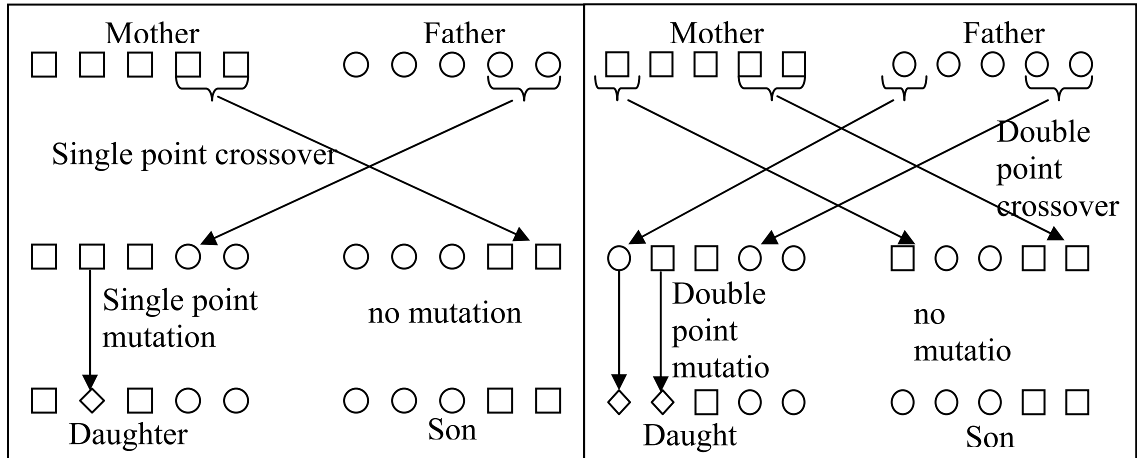


Figure III.2: Single and Double Point Recombination Operators

For a given problem, there might be different types of crossover and mutation operators, different method for setting the initial population and selection process. Additionally, a set of GA-related parameters need to be set properly, including population size, number of generations or stopping criteria, crossover and mutation rates. One disadvantage is the lack of a single combination of these techniques and parameters that will work best for all problems, or even different instances of the same problem. For example, a crossover operator that performs well for a particular problem may be ineffective for another problem, especially when the structures of the problems are significantly different. For this reason we have to perform extensive computational study in order to find good combination of the parameters. The main loop of GA is repeated until the stopping condition is satisfied. For example, a predefined number of generations or no change in the best solution for last “k” generations (where k is a predefined constant) can be two alternatives for stopping condition.

III.2. MEMETIC ALGORITMS

Although it is very easy to develop simple GA-based solutions for many optimization problems, they are not generally efficient for the complex combinatorial problems [24]. In order to improve performance of simple GAs, problem-specific knowledge, specialized operators or algorithms are incorporated with EAs to generate complex hybrid systems [10], which are called as *hybrid*

genetic algorithms, genetic local search algorithms, and *memetic algorithms*. Moscato [14] used the name memetic algorithm to include wide range of techniques where evolutionary-based search is augmented by the addition one or more phase of local search, or by the use of problem specific information. An online site [25] provides a comprehensive bibliography on memetic algorithms.

III.2.1. Structure of Memetic Algorithms

There are various ways that GA can be used with conjunction of other operators and/or problem specific knowledge. The four important ways of hybridization [10] are summarized below:

1. Heuristic can be added to generation of initial population. This will increase the efficiency of GA. Instead of randomly generate initial population previously known solutions arising from other techniques can be used to set the initial population. Additionally, local search method can be incorporated with the initial population generation phase in order to provide an initial population with a set of locally optimal points. In order to provide diversity, only a portion of the initial population is set or generated with the good solutions.
2. The crossover and mutation operators can be redesigned to represent problem-specific knowledge, instead of random crossover and mutation operators.
3. The most common use of hybridization is applying local search on whole solutions created by mutation or crossover. This can occur in different places before or after selection or after crossover and/or mutation. It is the application of improvement of individual members of the population during the life cycle. Local search moves one solution point to neighboring points to examine for an improved solution.
4. Prior to evaluation process a heuristic can be applied during the genotype–phenotype mapping. A heuristic about instance-specific knowledge can be used in the repair function.

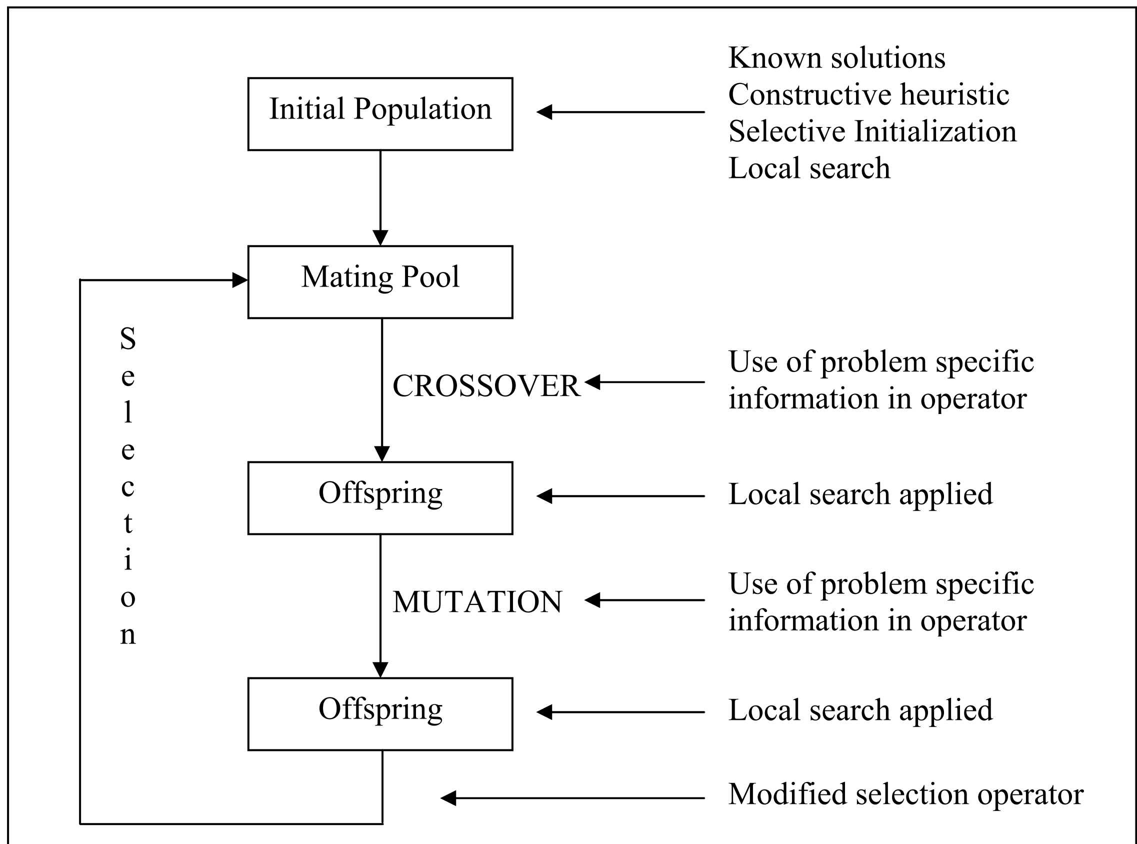


Figure III.3: Possible Places to Insert Problem Specific Knowledge and Other Operators in GA lifetime [10].

1. ***initialize*** population via local search
2. ***evaluate*** each candidate solution
3. ***while*** (termination criteria is not satisfied)
4.   ***select*** two solution as parents
5.   ***recombine*** parents
6.   ***mutate*** offspring
7.   ***evaluate*** new candidate solution
8.   ***improve*** new solution via local search
9.   ***select*** solutions for the next generation
10. ***end while***

Figure III.4: The Pseudocode for Hybrid GA [10]

Figure II.3 summarizes the places and methods of hybridization in Memetic Algorithms [10]. Figure III.4 gives a pseudocode for typical implementation of Hybrid GA [10]. The first step, generating initial population includes some heuristics and uses known solutions. Starting with good solution is an advantage, since it decreases the computational effort and increases the probability of finding

better solution. Then, local search can be applied to after generating initial population solutions by making the points in the solutions locally optimum. Local search is performed by selecting the neighbors of solutions wisely, calculating the fitness values of these neighbors, and choosing the better solution between them. Local search can be also performed on a set of or all individuals right after the crossover and mutation operators are performed.

III.2.1.1. Details on Local Search

It should be noted that there are various methods and algorithms that can be considered under the name local search. Local search is an iterative algorithm that moves from one solution *s* to another solution *s** according to some neighborhood structure. These algorithms are very generic and can be applied many problems successfully. Generally local search procedure has four steps [24]: initialization, neighbor generation, acceptance test and termination test. The difference between local search algorithms occurs at the acceptance test step. The other steps are same for all variants.

In the initialization step, an initial solution is chosen randomly or using heuristics as the current solution and the fitness value of this solution is calculated and it is represented as *f(s)*. In the second step, one of the neighbors of current solution is chosen and again the fitness value for chosen solution is calculated and it is represented as *f(s*)*. The acceptance test step differs the local search algorithms. In some algorithms only moves the better solutions are accepted; others also accept the moves to the worse solutions. The algorithm terminates after completing the predefined number of computation or computation time exceeds the predetermined limit. The important local search techniques are simulated annealing and hill climbing.

In simulated annealing, acceptance test known as probabilistic acceptance can be described as follows: (Δ term is computed by : $\Delta = f(s^*) - f(s)$)

- If $\Delta \leq 0$ then move to new location S* is always accepted.
- If $\Delta > 0$ then, it is moved to the S* with probability $e^{-\Delta/T}$ where T is a parameter called temperature. Usually T is large at the beginning and then it decreases until it is zero at final step.

In hill climbing a single solution is selected as the current solution and a new solution is selected from the neighborhood of the current solution at each iteration. If the new solution is better then it becomes the current solution. Otherwise some other neighbor is selected as the current solution. The method terminates if no further improvement is possible or for predefined number of iterations.

PART IV

PROPOSED FRAMEWORK FOR IMPROVING THE LOCALITY IN LOOPS

In this part, we present the details of our framework for improving the locality in imperfectly nested loops. The representation of loop nests, and statements in the framework are based on work proposed by Ahmed et al [1], which is presented in detail at section II.6.2.1. His work is considered as a reference in our framework, due to its representation of imperfectly nested loop. Section IV.1 demonstrates the major phases of our framework including the data structures considered, which is followed a section on details of hybrid genetic algorithm proposed.

IV.1. DETAILS OF THE FRAMEWORK

Our compiler optimization framework is not machine dependent since the generated code is not run on a machine and execution time of the program is not used as our fitness value of the solution. Our target is finding the optimal code for many machine architecture types by improving the data locality at source level. Additionally, it is a generic one that it is not tried to find good loop transformation parameters like tile size, unroll factors described at section II.2. We have tried to implement all types of loop transformations. The framework is not language dependent, since the parse part of the source code can be changed to any other

languages. The other parts of our optimization algorithm are not dependent to any language and we do not use any assumption on languages. Another important difference of our framework is that we process all statements and loop nests in a procedure not only one loop nest.

Figure IV.1 shows the flow diagram in our framework, which is also represented in terms of steps given at Figure IV.2. The details of the steps are presented at the following subsections of this part.

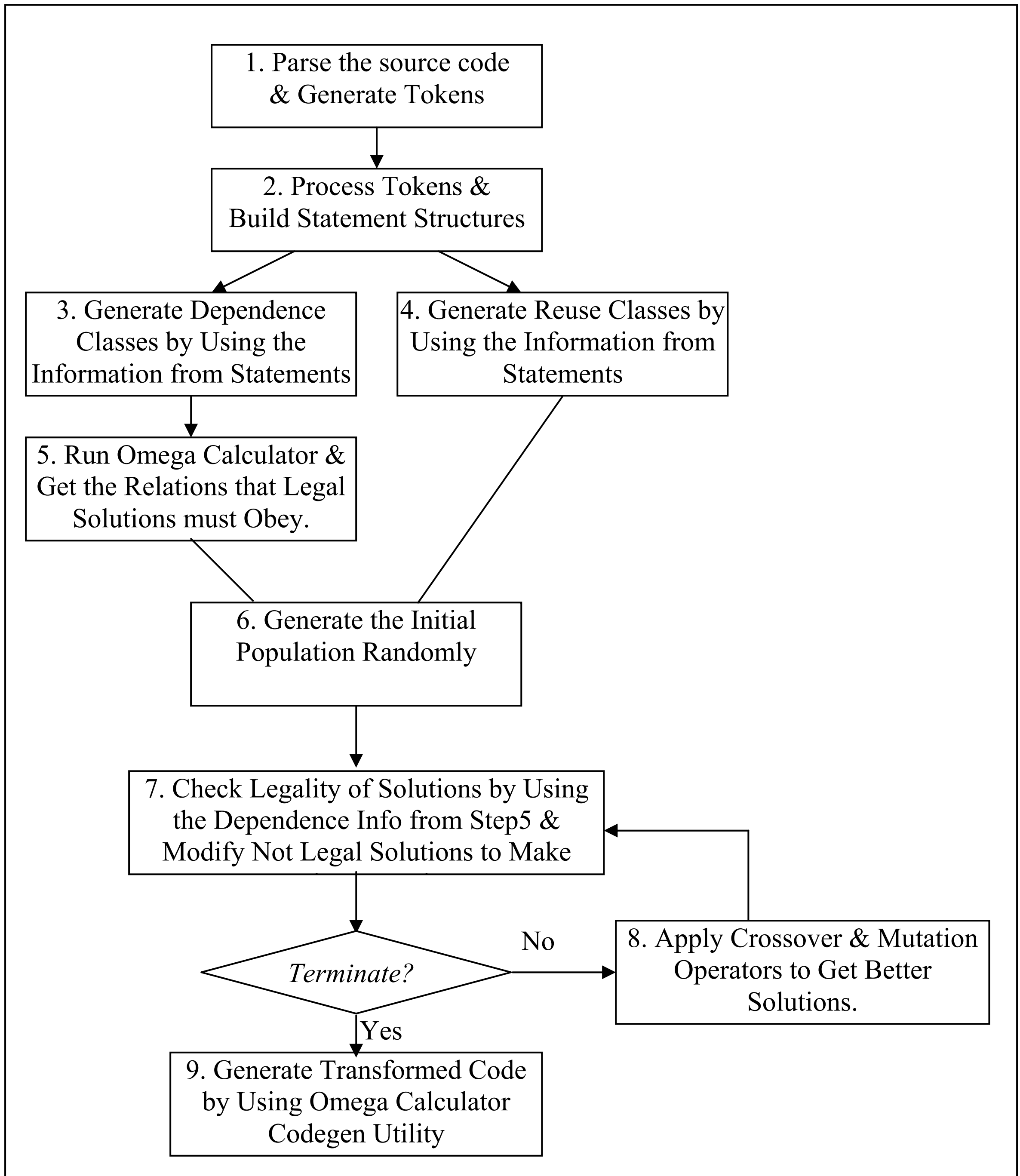


Figure IV.1: Flow Diagram of Our Optimization Framework

1. Parse Original Code & Produce Tokens
2. Generate Statement Structures by processing tokens
3. Find Dependencies between statements
4. Find Reuses between statements
5. Generate Constraints on embedding functions (mappings)
6. Generate Solutions
7. Check Legality of solutions
8. Find Best Solution by Using GA
9. Generate Transformed Code

Figure IV.2: The Steps of the Proposed Framework

The framework inputs a file that has the original code and it generates an output file that has the transformed code. The transformed code includes renamed iterations; since the original names of iterations are not preserved due to the transformations performed. The following example code (given in Figure IV. 3) is considered for explaining the phases of the framework throughout this part of thesis.

```
       for i = 1, M
           for j = 1, N
                for k = 1, j-1
S1:                  A(j,i) = A(j,i) – B(j,k)*A(k,i)
                end for
S2:             A(j,i) = A(j,i) / B(j,j)
           end for
       end for
```

Figure IV.3: An Example Code for Triangular Solve

IV.1.1. Parsing the original code into the tokens

We implemented a function for parsing a given input code. The delimiters like '[', ']', ' ', ',' are used to separate symbols, reserved keywords, etc. Token array includes also the meaningful delimiters like '[', ']' since they are used when constructing the statements and iterations structures. So token array has the elements that guide our while getting information about the statements. Figure IV.4 shows the first sixth element of the token array for the code sample at Figure IV.3.

tokenArr [0] = "***for***"	tokenArr [3] = "***1***"	tokenArr [6] = "***for***"
tokenArr [1] = "***i***"	tokenArr [4] = "***,***"	…………………..
tokenArr [2] = "="	tokenArr [5] = "***M***"	…………………..

Figure IV.4: Example Token Array Elements for the Sample Code.

IV.1.2. Generating the Statement Structures

The statement structures are constructed while processing the token array. The sequence is important in the token array. While tokens are being processed one by one, the iterations input and output array references per statement of the original code are constructed. Also at the same time, the structured information for input to the Omega Calculator is constructed like the loop bounds for new iterations of the product space. Figure IV.5 presents pseudocode of the statement generation phase.

```
1. initialize iteration_index and statement_index
2. if "for statement" is considered
        process "for statement":
                create an iteration instance
                set its lower, upper bounds and iterator symbols
                increase iteration_index
3. else if "assignment statement" is considered
        process "assignment statement":
                create a statement instance
                copy the iteration instances that this statement is inside of
                to new statement instance.
                increase statement_index
4. else do nothing just skip
```

Figure IV.5: Pseudocode for Construction of Iterations and Statements

The iteration_index and statement_index variables store the indexes for iterations and statements that show which statement and iteration is performed at the moment. Also these indexes are used to give numbers to statements and iterations. Iteration structure has information about the loops like lower bound, upper bound and iterator. The core structure of our algorithm is the statement structure. All

dependences, reuses and mappings are calculated by using info on the statements. Since all statements have their own iteration spaces in the product space, we copied the iterations info onto statements. The iterations are stored in the "IterationArr" element of the statements. After all we get the product space by combining the all statements iteration spaces. So we can obtain the dimensions of product space after processing the token array. Figure IV.6 shows the values of statements instances for the sample code at Figure IV.3.

```
Statement 1 :
        level           -> 3
        iteraions       -> i1,j1,k1
        iteration[0]    -> 1, M, i, i1
        iteration[1]    -> 1, N, j, j1
        iteration[2]    -> 1, j1-1 , j, k1
        bounds          -> 1 <= i1 <= M && 1 <= j1 <= N && 1 <= k1 <= j1-1
        output          -> A (j,i)
        input[0]        -> A (j,i)
        input[1]        -> B (j,k)
        input[2]        -> A (k,i)
```

```
Statement 2 :
        level           -> 2
        iteraions       -> i2,j2
        iteration[0]    -> 1, M, i, i2
        iteration[1]    -> 1, N, j, j2
        bounds          -> 1 <= i2 <= M && 1 <= j2 <= N
        output          -> A (j,i)
         input0         -> A (j,i)
         input1         -> B (j,j)
```

Figure IV.6: Values Stored at the Statement Structures for the example given at Figure IV.3.

IV.1.3. Finding Dependencies in the Statements

To check legality of solutions, the dependencies between the array references should be determined. Array references are stored with their keys and indexes in a structure. A statement has an output array reference and input array references as shown previous section. Storing the array references in this method is based on the set representation of dependencies mentioned at Section II.3.

We determine the dependencies by considering the definition given below. It should be noted that output dependence is false dependence since it can be eliminated by renaming and it is out of scope of our study.

Definition 1: There exist dependence between two statements, from source statement S_s to destination statement S_d if the following intersections have not empty set.

$$OUT(S_s) \cap IN(S_d) \neq \varnothing \quad \text{flow dependence}$$

$$IN(S_s) \cap OUT(S_d) \neq \varnothing \quad \text{anti dependence}$$

The dependence representation is important since we must check the legality of solutions across them. Polyhedral representation of dependencies is considered as in the Nawaaz Ahmed's work [1] described at section II.6.2.2.b. The intersection of the sets defined at Definition 1 is not enough for the loops. The loop bounds and array reference indexes are also important. We consider Definition 1 with the definition of dependencies in loops from Ahmed's work to find the dependencies in our algorithm.

Figure IV.7 shows the dependence as of the polyhedral form. Since $OUT(S_1) \cap IN(S_2)$ has not an empty set, there is a dependence from statement S_1 to statement S_2. Again $OUT(S_2) \cap IN(S_1)$ has not an empty set and second dependence exist from statement S_2 to statement S_1.

$$D_1 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$$

$$D_2 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, k_1 = j_2, i_1 = i_2 \}$$

Figure IV.7: Polyhedral Representations of Dependencies.

IV.1.4. Determining Reuses

Reuses represents the data locality in the code; and our algorithm find reuses in the similar way as in the dependences. In order to find dependencies it is required that one statement writes and other statement reads the same array reference. On the other hand, a reuse exists if two statements read or write to the same array reference. So we can define reuses based on the following definition.

Definition 2: There exist reuses between two statements, from source statement S_s to destination statement S_d if the following intersections are not empty set.

$$OUT(S_s) \cap OUT(S_d) \neq \varnothing$$

$$OUT(S_s) \cap IN(S_d) \neq \varnothing$$

$$IN(S_s) \cap IN(S_d) \neq \varnothing$$

$$IN(S_s) \cap OUT(S_d) \neq \varnothing$$

Our algorithm uses above definition to determine reuses between the statements. We represent the reuses in the polyhedral form by using the Ahmed's work given in detail at section II.6.2.2.c. Figure IV.8 shows the reuses in their polyhedral form.

$$R_1 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$$

$$R_2 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$$

$$R_3 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$$

$$R_4 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$$

$$R_5 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, k_1 = j_2, i_1 = i_2 \}$$

$$R_6 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, k_1 = j_2, i_1 = i_2 \}$$

$$R_7 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, k_1 = j_2 \}$$

Figure IV.8: Polyhedral Representations of Reuses.

There exist seven reuse classes from statement S1 to statement S2 for our sample code shown at Figure IV.3. First reuse exist since $OUT(S_1) \cap OUT(S_2)$ has not an empty set. Second reuse comes from $OUT(S_1) \cap IN(S_2)$, third and fifth ones from $IN(S_1) \cap OUT(S_2)$, fourth, sixth and seventh ones from $IN(S_1) \cap IN(S_2)$. Only forward directions are considered. The backward reuses, from S_2 to S_1, are not necessary since it has already considered at forward direction. It only generates the same set of reuses and multiplies the number of reuses by two.

IV.1.5. Generating Constraints on Unknown Coefficients of Mappings

The polyhedral form of dependence classes generated at step 3 of the proposed framework (see Figure IV.2) is used to find constraint on the mapping's unknown coefficients. We used a tool called Omega Calculator for this purpose. A brief explanation of the Omega Calculator tool is given at Appendix A. Finding constraints is not a simple job. Until now we determine the dependencies. The mappings, which are candidate solutions, must obey the rules from these dependencies. Firstly we must understand that what dependence relations represent. Since we combine the statement iteration spaces, dependence relations guide us that how they will be combined to form a valid product space. In other words, it shows the relation between the indices of the combined iteration spaces hence the rules from dependencies are not violated. For instance, there exist two dependencies for the sample code shown at Figure IV.3, which are:

$D_1 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \le i_1, i_2 \le M, 1 \le j_1, j_2 \le N, 1 \le k_1 \le j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$

$D_2 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \le i_1, i_2 \le M, 1 \le j_1, j_2 \le N, 1 \le k_1 \le j_1\text{-}1, k_1 = j_2, i_1 = i_2 \}$

These two dependence classes form two polyhedrons that are constructed from the relations above. Transformed code iterations must be inside of this polyhedron. These polyhedrons can be described as the affine functions. The affine form of D_1 is given Figure IV.9.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i_1 \\ j_1 \\ k_1 \\ i_2 \\ j_2 \\ M \\ N \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ 0 \\ 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \ge \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Figure IV.9: Affine representation of first dependence in the example.

The Farkas Lemma [26] proves that a function is always has a positive value on a polyhedron domain, which is given in detail at Appendix B. We can obtain the constraints on the product space dimension by applying the Farkas Lemma [26] to the affine form of dependencies. The f(x) functions come from the difference vectors of dependencies and all solutions have their own dependence difference vectors. But the right side of the equation comes from the dependence classes. Since the dependence classes are generated from the source code at once, the all constraints on the product space dimensions generated only once.

Several polyhedral libraries were considered and it was decided to use Omega Calculator**.** The Omega Calculator has a function (called *Farkas*), which implements the Farkas Lemma. Also it is very useful since there is no need to convert dependence to the affine form; which the Omega Calculator can do. Since Omega Calculator requires a file as its input and returns the result in a file, then all dependencies should be written to a file in the form of Omega Calculator; and it is executed once. Then, it generates an output file and this file must be parsed to get the resulting relations. Figure IV.10 gives an example to a sample input file format to Omega Calculator and the resulting output file format.

```
symbolic n;
R1 := { [i1,j1,i2,j2,k2] : 1 <= i1,j1,i2,j2,k2 <= n && i1=i2 && j1=j2};
R2 := farkas R1;
R2;
```

```
# Omega Calculator v1.2 (based on Omega Library 1.2, August, 2000):
# symbolic n;
#
# R1 := { [i1,j1,i2,j2,k2] : 1 <= i1,j1,i2,j2,k2 <= n && i1=i2 && j1=j2};
#
# R2 := farkas R1;
#
# R2;

{[i1,j1,i2,j2,k2]: 0 <= n && 0 <= n+k2 && 0 <= n+i1+i2+k2 && 0 <= n+i1+i2
&& 0 <= n+i1+j1+i2+j2+k2 && 0 <= n+i1+j1+i2+j2 && 0 <=
constantTerm+n+i1+j1+i2+j2+k2 && 0 <= n+j1+j2 && 0 <= n+j1+j2+k2}

#
```

Figure IV.10: (a) Content of the Input File to Omega Calculator (b) Content of the Output File from Omega Calculator

While Omega Calculator applies Farkas lemma for the given relations, it also implements the Fourier Motzkin elimination to eliminate Farkas multipliers. The steps of Fourier Motzkin elimination is explained at Appendix C. You can see how a relation can be defined in omega calculator form at Figure IV.10 (a). Applying Farkas lemma gives the result shown at Figure IV.10 (b). There exist nine relations in this output. These relations give us the constraints on unknown coefficients of the mappings. For example second relation "0 <= n + k2" tells us the coefficient of the term "n" + the coefficient of the term "k2" must be grater or equal to zero to provide input relation of the "farkas" function.

IV.1.6. Checking the Legality of Mappings

In this section we present how we will control that the semantic of the program remains unchanged. For a set of mappings $\{F_1, F_2, \dots Fn\}$, walking the product space lexicographically and executing the all statement instances (mapped to each point as we visit it) will generate the execution order of the statements. To obtain the legal order, the transformed product space must satisfy all dependencies. The difference vector for a dependence class D, which dependence exists from statement 1 to statement 2, is defined as follows [1]:

$$V(i_s,i_d) = F_d\,(i_d) - F_s(i_s)$$

Then we can say that the set of mappings $\{F_1, F_2, \dots Fn\}$ is valid if the all entries of the all difference vectors of the program are zero or positive values. At the previous section we mentioned that the constraints comes from the dependencies are calculated by applying Farkas Lemma. To check legality of the embeddings we use these constraints. Now the f(x) function that is mentioned at Farkas Lemma is our difference vectors. We want to difference vector non-negative everywhere in the domain of polyhedron from dependence relations. After applying Farkas Lemma, we get some affine equations, which include the product space dimensions. We must look at the difference vector whether satisfies these inequalities or not.

Figure IV.11 shows an example one valid solution and one invalid solution according to the relation from Farkas Lemma. Let's assume we get an inequality after applying Farkas Lemma like $i_1+i_2+ConstantTerm \geq 0$. This inequality tells us that the sum of coefficients of i_1, i_2 and constant term must be equal or greater than zero. At Figure IV.11.a, the first dimension of difference vector violates this rule.

The coefficients of i_1, i_2, constant term are –1, 1 and –1, respectively. Since the sum of coefficients is –1 this solution is invalid. Figure IV.11 (b) shows a valid solution according to the given inequality. Now the coefficients are –1, 1 and 0 at the first dimension and –1, 1 and 1 at the fourth dimension. The sum of the coefficients is 0 and 1, respectively. So this solution is valid base on the given inequality. At real applications a lot of inequalities comes from the dependencies that seem the inequality given in this example.

$$F_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} ; \; F_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_2 \\ j_2 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

$$F_2 - F_1 = \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \\ i_2 \\ j_2 \\ M \\ N \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \\ -2 \\ 1 \\ 0 \end{bmatrix}$$

(a) Invalid Solution Mappings

$$F_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} ; \; F_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_2 \\ j_2 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix}$$

$$F_2 - F_1 = \begin{pmatrix} -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \\ i_2 \\ j_2 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -2 \\ 1 \\ 0 \end{bmatrix}$$

(b) Valid Solution Mappings

Figure IV.11: Valid and Invalid Solutions for the example given at Figure IV.3.

IV.1.7. Generating Transformed Code from Mappings

Our algorithm uses the Omega Calculator "codegen" function again to generate transformed code from the found best mapping. At the last step of algorithm Omega Calculator runs one more time to generate output code. Since the Omega Calculator accepts an input file, the best solution mappings are written to a file in its format. After run Omega Calculator the result file for our algorithm that includes the transformed code generated. Figure IV.12 gives an example transformed code and the mapping that yields this transformation. Also the input file format and output file format of the Omega Calculator is shown at Figure IV.12.

```
T10:={[t,i,j] -> [t,j,i]};
T20:={[t,i,j] -> [t,j+1,i+1]};
Symbolic T,N;
IS10:={[t,i,j] : 1<=t<=T && 2<=i,j<=N-1};
IS20:={[t,i,j] : 1<=t<=T && 2<=i,j<=N-1};
codegen T10:IS10,T20:IS20;
```

```
if (N >= 3) {
  for(t1 = 1; t1 <= T; t1++) {
    for(t2 = 2; t2 <= N; t2++) {
      if (N >= t2+1) {
        s1(t1,2,t2);
      }
      if (t2 <= 2) {
        for(t3 = 3; t3 <= N-1; t3++) {
          s1(t1,t3,2);
        }
      }
      if (N >= t2+1 && t2 >= 3) {
        for(t3 = 3; t3 <= N-1; t3++) {
          s1(t1,t3,t2);
          s2(t1,t3-1,t2-1);
        }
      }
      if (t2 >= 3 && N >= t2+1) {
        s2(t1,N-1,t2-1);
      }
      if (N <= t2) {
        for(t3 = 3; t3 <= N; t3++) {
          s2(t1,t3-1,N-1);
        }
      }
    }
  }
}
```

Figure IV.12: An Example for the Omega Calculator Code Generation Facility, and the Mappings that Causes This Transformation.

IV.2. HYBRID GENETIC ALGORITHM OF THE FRAMEWORK

We propose a steady state genetic algorithm that takes two parents and generates one offspring by recombination operators. The hybridization is performed first for setting initial population by considering a problem specific heuristic. The second hybridization is performed to improve the offspring before inserting it into the population. In this subsection we present the details of our hybrid genetic algorithm.

IV.2.1. Solution Representation

As mentioned before our aim is finding the good transformation for a source code that provides most efficient data locality. The transformation is constructed from mappings for each statement of the given source code. Then our goal is to find the best mappings, which maps statements iteration space to the product space. As a result our solution includes mappings of each statement. Figure IV.13 shows the affine form of mappings as our solution representations.

$$F_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_1 \\ j_1 \\ k_1 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} i_1 \\ j_1 \\ k_1+1 \\ i_1 -1 \\ j_1 \end{bmatrix}, \quad F_1 = A_1 x_1 + b_1$$

$$F_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{bmatrix} i_2 \\ j_2 \\ M \\ N \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} i_2 \\ j_2 \\ j_2-1 \\ i_2 \\ j_2 \end{bmatrix}, \quad F_2 = A_2 x_2 + b_2$$

Figure IV.13: Solution Representation of Mappings in Their Affine Form

In our algorithm a solution has all mappings for all statements. In the example shown at Figure IV.13, there exist two mappings since the sample code given at Figure IV.3 has two statements. First statement iteration space has three dimensions as {i1, j1 and k1}, and second one has two dimensions as {i2 and j2}.

So the product space has five dimensions; and all mappings map their iterations spaces to a five dimensional product space. So the number of rows is equal to product space dimension and the number of columns is equal to (product space dimension + the number of symbols + 1) which the last entry is added as a constant term.

IV.2.2. Initial Population Generation

In the initial population, predefined numbers of solutions are generated randomly by using the problem specific information. As we mentioned at previous section, our solution is the statement mappings. The mappings have two parts a constant matrix and a constant vector. The matrix and vector for all statement mappings are generated randomly.

```
1.  initialize all mappings of all solutions.
2.  while solution_counter < population size
3.      while mapping_counter < statement count
4.          while dimension_couter < product space dimension
5.              if  decide to choose a dimension
6.                  select one dimension among the iteration space dimensions
7.                  assign this dimension entry of the mapping 1.
8.              end if
9.              if  decide to choose a symbol
10.                 select one  of the symbol
11.                 assign this symbol entry of the mapping 1.
12.             end if
13.             if  decide to choose constant
14.                 get a random number between predefined range
15.                 assign the constant term entry of the mapping this number
16.             end if
17.             increase dimension_counter
18.         end while
19.         increase mapping_counter
20.     end while
21.     check legality of solutions according to the dependencies
22.     if not legal
23.         make it legal by arranging the mappings
24.         if  legal
25.            increase solution_counter
26.         end if
27.     end if
28. end while
```

Figure IV.14: Our Initial Population Generation Algorithm.

There are two main considerations that are very important for generating valid mappings. A mapping cannot be legal if the entries of the matrix A have the values other then 1, -1 and 0. Moreover a mapping, which maps i1 dimension to i1+j1, cannot be legal. Figure IV.14 shows the algorithm for the initial population generation. While we are generating initial population, we check whether the solution is valid or not. The dependence constraints come from output of Farkas lemma implementation is used to decide the solution is legal or not. If an invalid solution is generated in initial phase, it is tried to be converted to a valid solution for a predefined number of iterations. If it is not successful to validate the solution, it is discarded.

IV.2.3. Recombination Operators

A single point crossover operator combined with problem specific knowledge is considered in our GA-based solution. The crossover operator can be applied to the mappings belonging to the same statements; and application of crossover operator between the different statements mapping generates invalid solutions, since all statements have their own iteration space.

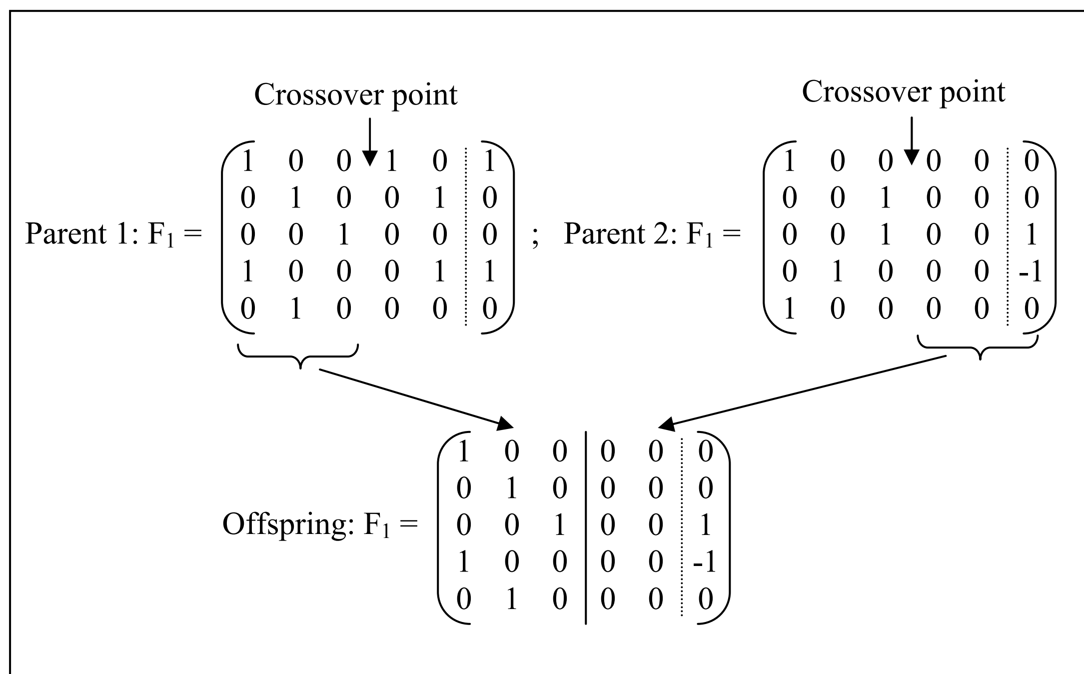


Figure IV.15: Example Crossover Operation with Crossover Point 3 over the First Statement Mappings.

We chose two parents with the Roulette wheel selection method and we choose the mapping to which crossover applied. After finding the mapping, the crossover point is determined. Only one offspring is produced from these parents. Figure IV.15 shows an example to crossover operation when the crossover point is selected as 3, randomly. The mappings at Figure IV.15 we combine the matrix A and vector a, for simplicity.

We implemented single point mutation operation. Our mutation operator firstly decides whether to mutate the new offspring or not. If mutation will be done, it randomly chooses which dimension and which column to mutate. Mutation is simply done by making 0 entries to 1 and 1 entries to 0. Figure IV.16 shows an example to mutation operation on the offspring generated at Figure IV.15. The fourth dimension and third column is selected to mutate. Since this entry has a zero value, mutation make it 1.

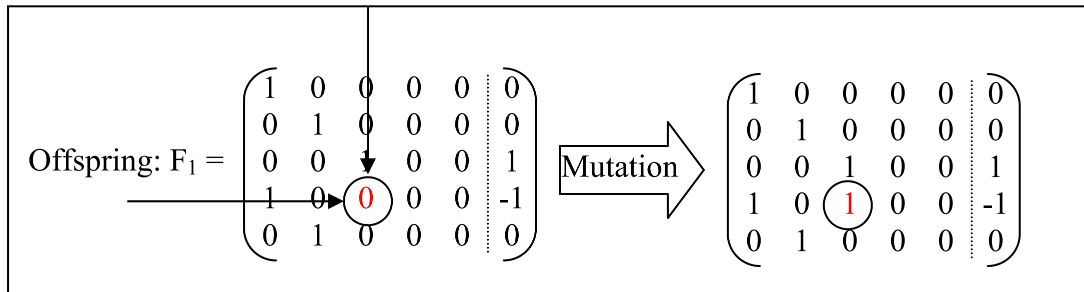


Figure IV.16: Example Mutation Operation with Points (4,3).

The new offspring must be checked against the dependencies due to the possibility of invalid mappings. Therefore the arrangement after the generation of offspring must be applied. The rearrangement process is applied for the predefined times to get valid solutions. If a valid solution cannot be obtained at the end, this solution is discarded and regeneration starts again. Discarded solutions are not counted as a generation and not increments the generation counter.

IV.2.4. Local Search

Our algorithm implements the local search method at initial population generation and after recombination operators generate a new offspring. Local search is applied to improve solution by checking a set of its neighbors. A neighbor of a

given solution is obtained by randomly choosing a row and column to update. A modification on the matrix part is done by making 1's to 0's and 0's to 1's to generate the solution neighbors. It can generate a not valid solution again we try to make the neighbor solution valid. Our goal is finding better combination for a given solution in the neighbor space. We implemented the hill-climbing algorithm for the local search phase. In this section we give the details of this method on our problem.

$$\text{Offspring: } F_1 = \left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$\text{Neighbor Offspring: } F_1 = \left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & \textcircled{1} \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & \textcircled{1} & \textcircled{0} & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right)$$

Figure IV.17: An example offspring and its neighbor

If we define the fitness values of offspring as *f(s)* and neighbor offspring as *f(s*)*, the local search described in the hill-climbing method, acts as follow:

- If $f(s^*) - f(s) \geq 0$ then update the offspring *s* from its neighbor *s**.
- If $f(s^*) - f(s) < 0$ then choose another offspring *s* at random.

Our problem is maximization; so offspring *s** is better than offspring *s* if $f(s^*) - f(s)$ is greater than zero. Searching the neighbors continues until no improvement done for the last k iterations. At the end the last selected solution is returned as a new offspring to be joined to population.

IV.2.5. Fitness Function Evaluation

There exist three level of fitness function evaluations performed in our framework. These are dimension based, mapping based and solution based fitness function. All of them are shown at Figure IV.18. As we mentioned before measuring the cache misses or executing the transformed code and getting the

execution time is not practical, it takes a lot of time. Especially GA generates many valid solutions and calculated the fitness value for each of them. So calculation of fitness value must not require too much time. Also these techniques for fitness value calculation make the algorithm machine dependent. If we used these techniques, it is required that the code must be compiled for the specific machine. To have general fitness value we used the Ahmed's [10] approach of measurement of goodness of mappings. He suggests usage of reuses to measure the improvement. Since his algorithm determines mappings dimension by dimension, he does not consider all reuses while determining mappings. Our algorithm also differs at this point. We consider all reuses while calculating the fitness value. Since our aim is increasing the data locality our fitness function is based on the reuses.

$$F_1 = \left(\begin{array}{ccccc:c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \begin{array}{c} \text{Fitness} \\ 6 \\ 30 \\ 24 \\ 0 \\ 24 \end{array} \qquad F_2 = \left(\begin{array}{cccc:c} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right) \begin{array}{c} \text{Fitness} \\ 6 \\ 30 \\ 24 \\ 0 \\ 24 \end{array}$$

Fitness $F_1 = 84$ Fitness $F_2 = 84$

Solution Fitness $F = 168$

Figure IV.18: Fitness Values for the example given at Figure IV.3

We have already mentioned how the reuses in the code is found and represented in polyhedral form at section IV.1.4. For each reuses we construct the difference vector and for each dimension in this difference vector we look at the entry whether it is zero or non-zero according to the given reuse. If it is zero then we give a reward to this dimension and increment its corresponding dimension fitness values in the mapping by one. After all reuses are controlled in this way we finish the dimension based fitness values of solutions. Mapping based fitness values are calculated by summing the dimension fitness values of them. At the end solution based fitness values are also the sum of all mappings fitness values.

PART V

EXPERIMENTAL STUDY

In this section we present the experimental study to measure the effectiveness of our framework. The first part of testing is dedicated to set Genetic algorithm parameters, which include population size (P), number of generations (G), crossover probability (p_c) and mutation probability (m_c).

The performance of our framework is examined when the population size is in the range P={50, 100, 150, 200, 250} for the chosen sample codes. It was observed that the performance of the algorithm does not depend on population size when the number of iteration is fixed. The best results in terms of transformed code performance are observed with the lowest population size, i.e., when P is equal to 50. This is due to the fact that the population of individuals is not completely generated in every generation. Additionally, the proposed algorithm for the initial population (given in Section IV.2.2.) generates a limited number of different individuals because of dependence constraints; and when the population size is increased, there will be various replications of individuals in the population.

We have also examined the best c_p-m_p pairs; we have used the c_p values in the range {0.5, 0.6, 0.7, 0.8, 0.9, 1.0} and m_p values in the range {0.10, 0.15, 0.20, 0.25, 0.30}. The crossover probability is usually given a high probability. On the other hand, mutation can be given a relatively small probability. At the end, we have found the best parameter set of c_p=0.8, and m_p=0.2. The tests are done with the population size of 50 and for 250 generations. At low crossover probability the success probability of generating best solution is decreased. On the other hand at very high crossover probability it is increased with increasing running time. The

same success rate is also obtained when crossover probability is equal to 0.8. Again high values of mutation probability is increased the execution time of framework since the framework consumes much more time to make solutions valid after mutation operators generates invalid solutions. On the other hand the smallest mutation probabilities is decreased the variations of solutions and decreased the success rate of our genetic algorithm to find best solution.

After each parameter is set with the best value that is observed, the main experiments are performed on four important codes, Triangular Solve with multiple left-hand sides, Cholesky Factorization, Jacobi Kernel and Red-Black Gauss-Seidel. All experiments we run in he domain called "moonstar" at Istanbul Technical University. It has 12 CPU (900 Mhz UltraSPARC III-Cu) with 24 GB memory. We present the following performance numbers for each source code (SC) and transformed code (TC).

1. Performance of code produced by Sun compiler
2. Performance of code produced by Sun compiler with optimization
3. Performance of code produced by GNU compiler
4. Performance of code produced by GNU compiler with optimization

GNU compiler –O3 flag is turned on to allow its optimization so we can compare the result with our framework optimization. For the same reason, -xO5 flag of the SUN compiler is turned on. The number behind the –O is called the optimization level. The default is O0, which means do not optimize. More details about optimization flags are given at [32, 33]. The source code and transformed code are compiled and run for the various sized matrixes for 5 times for all. The average execution times are obtained and comparison made between the transformed code and the original one. The performance numbers shows the benefits of finding better transformations instead of searching the best transformation sequence sequentially. Even though the Sun compiler and GNU compiler implements all the transformations necessary to optimize our benchmarks, they do not find the right sequence of transformations for source code, so the performance of resulting code suffers.

V.1. PERFORMANCES

We represent our framework results on four important code samples in the subsections.

IV.1.1. Triangular Solve

The original source code and the transformed code generated by our algorithm for triangular solve with multiple right-hand sides shown at Figure V.1.

```
    for c = 1, M
        for r = 1, N
            for k = 1, r-1
S1:              B(r,c) = B(r,c) - L(r,k) * B(k,c)
            end for
S2:         B(r,c) = B(r,c) / L(r,r)
        end for
    end for
```

```
    for c = 1, M
        for r = 1, N
S2:          B(r,c) = B(r,c) / L(r,r)
            for k = r+1, N
S1:              B(k,c) = B(k,c) - L(k,r) * B(r,c)
            end for
        end for
    end for
```

Figure V.1: (a) Original Code and (b) Transformed Code for Triangular Solve

The embeddings generated by our framework shown at Figure V.2. Although the framework find five dimensional spaces, the omega calculator gives us the generated code as at most three levels as shown at FigureV.1 (a). It removes the redundant dimensions for us.

$$F_1 \begin{pmatrix} c_1 \\ r_1 \\ k_1 \end{pmatrix} = \begin{pmatrix} c_1 \\ k_1 \\ r_1 \\ c_1 \\ r_1 \end{pmatrix} \qquad F_2 \begin{pmatrix} c_2 \\ r_2 \end{pmatrix} = \begin{pmatrix} c_2 \\ r_2 \\ r_2 \\ c_2 \\ r_2 \end{pmatrix}$$

Figure V.2: Generated Mappings for Two Statements of Triangular Solve

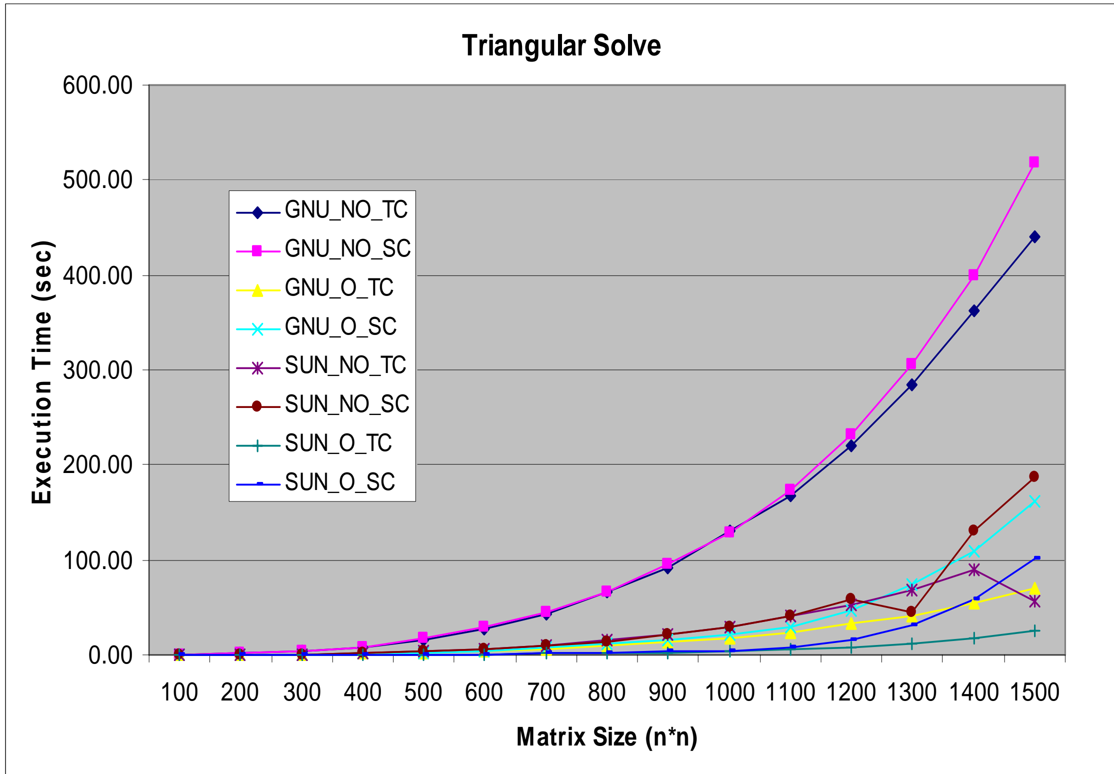


Figure V.3: Graphical Performances for Triangular Solve

Matrix Size(n*n)	GNU				SUN			
	Not Optimized		Optimized (-O3)		Not Optimized		Optimized (-xO5)	
	TC	SC	TC	SC	TC	SC	TC	SC
100	0.13	0.12	0.02	0.02	0.03	0.03	**0.00**	0.00
200	0.99	1.15	0.13	0.16	0.22	0.22	**0.02**	0.03
300	3.37	3.63	0.57	0.56	0.77	0.75	**0.09**	0.10
400	8.02	8.67	1.07	1.32	1.82	1.80	**0.22**	0.25
500	15.59	16.83	2.25	2.59	3.56	3.51	**0.42**	0.49
600	27.52	29.21	3.77	4.50	6.17	6.10	**0.73**	0.90
700	43.13	44.64	6.06	7.15	9.87	9.72	**1.16**	1.41
800	65.64	66.99	9.34	10.81	14.81	14.43	**1.74**	2.14
900	92.18	95.90	12.97	15.47	21.07	21.02	**2.53**	3.14
1000	130.14	128.44	17.95	21.42	29.32	28.73	**3.42**	4.50
1100	168.43	174.27	24.13	29.68	40.29	40.55	**4.97**	7.31
1200	219.66	231.54	32.93	46.21	51.87	58.87	**7.45**	15.57
1300	283.83	306.57	41.32	73.93	67.47	44.84	**12.45**	31.78
1400	361.50	399.12	54.90	108.97	88.92	129.99	**18.18**	59.32
1500	440.21	517.60	70.75	162.38	121.86	186.80	**24.97**	100.77

Table V.1: Numerical Performances for Triangular Solve

Figure V.3 and Table V.1 shows the execution times for original source code and transformed code for various compiler and settings. Approximately in all cases the transformed code performance is the best. If we look the values of the case transformed code with SUN optimization, we can see it has the best performance among all cases. Our framework does not consider the tiling. It can be another study on our framework. The SUN compiler optimization gives better result by applying the tilling to our code. On the other hand we can say that the codes produced by our framework is more proper to optimize by compilers. If we compare two commercial compilers we can say that the SUN compiler is the best on SUN server since it knows the architectural details and make the best optimization on known architecture.

IV.1.2. Red-Black Gauss-Seidel Relaxation

The original source code and the transformed code generated by our algorithm for Jacobi shown at Figure V.4.

```
   for t = 1, T
      for j = 2, N-1
         for i = 2, N-1, 2
S1:         U(i,j) = 0.25*(B(i,j)-U(i-1,j)-U(i+1,j)-U(i,j+1)-U(i,j-1))
         end for
      end for
      for i = 2, N-1
         for j = 3, N-1, 2
S2:         U(i,j) = 0.25*(B(i,j)-U(i-1,j)-U(i+1,j)-U(i,j+1)-U(i,j-1))
         end for
      end for
   end for
```

```
   for t = 1, T
      for j = 2, N-1
S1:       U(2,j)=0.25*(B(2,j)-U(1,j)-U(3,j)-U(2,j+1)-U(2,j-1));
         for i = 4, N-1, 2
S2:         U(i,j)=0.25*(B(i,j)-U(i-1,j)-U(i+1,j)-U(i,j+1)-U(i,j-1))
S3:         U(i-2,j)=0.25*(B(i-2,j)-U(i-3,j)-U(i-1,j)-U(i-2,j+1)-U(i-2,j-1))
         end for
S4:       U(N-2,j)=0.25*(B(N-2,j)-U(N-3,j)-U(N-1,j)-U(N-2,j+1)-U(N-2,j-1))
      end for
   end for
```

Figure V.4: (a) Original Code and (b) Transformed Code for Red-Black Gauss-Seidel Relaxation

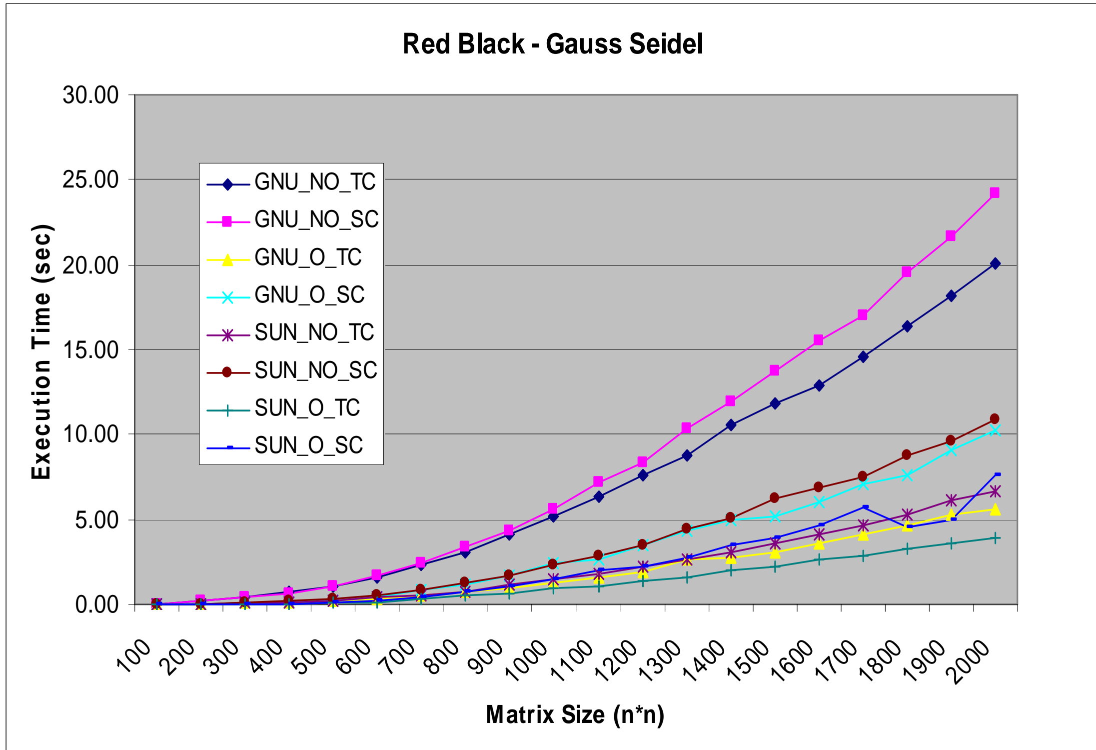


Figure V.5: Graphical Performances for Red-Black Gauss-Seidel

Matrix Size(n*n)	GNU				SUN			
	Not Optimized		Optimized (-O3)		Not Optimized		Optimized (-xO5)	
	TC	SC	TC	SC	TC	SC	TC	SC
100	0.04	0.04	0.01	0.01	0.01	0.01	**0.00**	0.00
200	0.17	0.17	0.03	0.04	0.03	0.04	**0.01**	0.01
300	0.38	0.39	0.07	0.09	0.08	0.10	**0.03**	0.03
400	0.69	0.68	0.12	0.16	0.14	0.19	**0.05**	0.05
500	1.10	1.08	0.18	0.24	0.25	0.28	**0.07**	0.08
600	1.61	1.69	0.32	0.47	0.39	0.53	**0.16**	0.22
700	2.33	2.44	0.51	0.82	0.56	0.86	**0.28**	0.43
800	3.09	3.36	0.71	1.20	0.78	1.25	**0.52**	0.72
900	4.10	4.37	0.96	1.72	1.15	1.74	**0.65**	1.06
1000	5.17	5.60	1.24	2.43	1.45	2.31	**0.90**	1.45
1100	6.31	7.16	1.59	2.69	1.79	2.88	**1.09**	1.96
1200	7.60	8.32	1.95	3.54	2.17	3.49	**1.33**	2.23
1300	8.80	10.31	2.61	4.36	2.60	4.43	**1.56**	2.79
1400	10.52	11.97	2.76	4.92	3.02	5.09	**1.97**	3.46
1500	11.79	13.76	3.08	5.17	3.55	6.26	**2.18**	3.90
1600	12.89	15.49	3.54	5.99	4.10	6.85	**2.61**	4.65
1700	14.59	17.00	4.09	7.08	4.68	7.46	**2.89**	5.71
1800	16.37	19.51	4.61	7.66	5.29	8.72	**3.29**	4.55
1900	18.12	21.69	5.23	9.05	6.09	9.63	**3.59**	4.92
2000	20.03	24.19	5.56	10.29	6.62	10.83	**3.87**	7.58

Table V.2: Numerical Performances for Red-Black Gauss-Seidel

Figure V.5 shows the results for Red-Black Gauss-Seidel code. It is clear that in all cases the transformed code by our framework has the minimum execution times. As in the previous sample code the best result is obtained by combining our optimization with compilers one. Our framework generates more suitable codes to optimize for compilers. Our framework chooses the embeddings shown at Figure V.6 to optimize the source code:

$$F_1 \begin{pmatrix} t_1 \\ j_1 \\ i_1 \end{pmatrix} = \begin{pmatrix} t_1 \\ j_1 \\ i_1 \\ t_1 \\ j_1 \\ i_1 \end{pmatrix} \qquad F_2 \begin{pmatrix} t_2 \\ j_2 \\ i_2 \end{pmatrix} = \begin{pmatrix} t_2 \\ j_2 \\ i_2+1 \\ t_2 \\ j_2 \\ i_2+1 \end{pmatrix}$$

Figure V.6: Generated Mappings for Two Statements of Red-Black Gauss-Seidel

IV.1.3. Jacobi

The original source code and the transformed code generated by our algorithm for Jacobi shown at Figure V.7.

```
   for t = 1, T
      for i = 2, N-1
         for j = 2, N-1
S1:         L(i,j) = (A(i,j+1) + A(i,j-1) + A(i+1,j) + A(i-1,j)) / 4
         end for
      end for
      for i = 2, N-1
         for j = 2, N-1
S2:         A(i,j) = L(i,j)
         end for
      end for
   end for
```

```
   for t = 1, T
S1:   L(2,2) = (A(2,3) + A(2,1) + A(3,2) + A(1,2))/4
      for j = 3, N-1
S2:      L(2,j) = (A(2,j+1) + A(2,j-1) + A(3,j) + A(1,j))/4
      end for
      for i = 3, N-1
S3:      L(i,2) = (A(i,3)+A(i,1)+A(i+1,2)+A(i-1,2))/4
         for j = 3, N-1
S4:         L(i,j) = (A(i,j+1) + A(i,j-1) + A(i+1,j) + A(i-1,j))/4
S5:         A(i-1,j-1) = L(i-1,j-1)
         end for
S6:      A(i-1,N-1) = L(i-1,N-1)
      end for
      for j = 3, N
S7:      A(N-1,j-1) = L(N-1,j-1)
      end for
   end for
```

Figure V.7: (a) Original Code and (b) Transformed Code for Jacobi

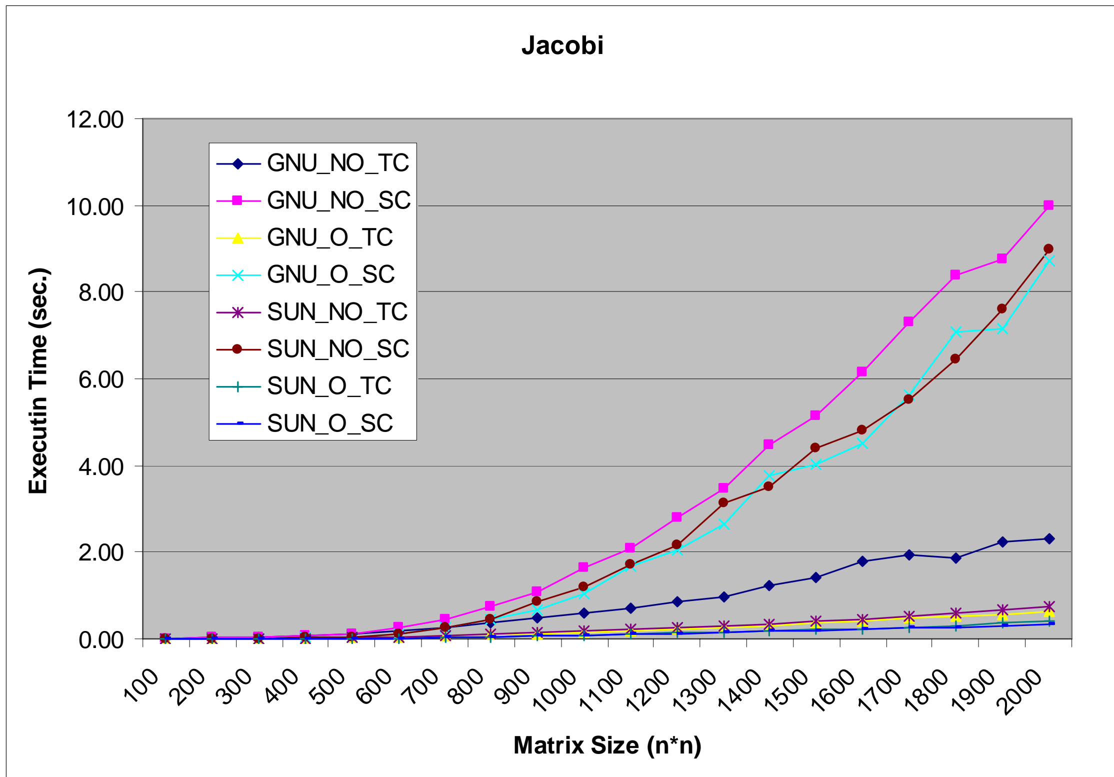


Figure V.8: Graphical Performances for Jacobi

Matrix Size(n*n)	GNU				SUN			
	Not Optimized		Optimized (-O3)		Not Optimized		Optimized (-xO5)	
	TC	SC	TC	SC	TC	SC	TC	SC
100	0.01	0.01	0.00	0.00	0.00	0.00	**0.00**	**0.00**
200	0.02	0.02	0.00	0.01	0.00	0.01	**0.00**	**0.00**
300	0.05	0.05	0.01	0.01	0.01	0.01	**0.00**	**0.00**
400	0.08	0.09	0.01	0.02	0.02	0.02	**0.01**	**0.00**
500	0.13	0.13	0.03	0.04	0.03	0.03	**0.01**	**0.00**
600	0.19	0.26	0.04	0.12	0.05	0.12	**0.02**	**0.01**
700	0.27	0.45	0.07	0.25	0.07	0.25	**0.03**	**0.02**
800	0.38	0.74	0.09	0.43	0.11	0.45	**0.05**	**0.04**
900	0.47	1.06	0.12	0.69	0.14	0.87	**0.06**	**0.06**
1000	0.58	1.63	0.15	1.04	0.17	1.21	0.09	**0.08**
1100	0.70	2.08	0.19	1.66	0.21	1.70	0.11	**0.10**
1200	0.85	2.78	0.23	2.07	0.26	2.15	0.14	**0.11**
1300	0.99	3.46	0.27	2.65	0.30	3.12	0.16	**0.14**
1400	1.22	4.47	0.31	3.75	0.35	3.52	0.18	**0.18**
1500	1.42	5.16	0.36	4.02	0.39	4.41	0.22	**0.20**
1600	1.78	6.15	0.42	4.52	0.46	4.80	0.24	**0.21**
1700	1.95	7.30	0.47	5.64	0.53	5.51	0.28	**0.26**
1800	1.87	8.37	0.52	7.10	0.60	6.44	0.30	**0.28**
1900	2.24	8.77	0.57	7.14	0.68	7.60	0.37	**0.31**
2000	2.31	9.97	0.64	8.73	0.73	8.97	0.41	**0.34**

Table V.3: Numerical Performances for Jacobi

Figure V.8 shows the performances. In this case the transformed code and source code performance are approximately same. We think for the Jacobi code the sun compiler optimizer make almost same transformations with our framework. The embeddings chosen by our framework for Jacobi is shown at FigureV.9.

$$F_1 \begin{pmatrix} t_1 \\ i_1 \\ j_1 \end{pmatrix} = \begin{pmatrix} t_1 \\ j_1 \\ i_1 \\ t_1 \\ j_1 \\ i_1 \end{pmatrix} \qquad F_2 \begin{pmatrix} t_2 \\ i_2 \\ j_2 \end{pmatrix} = \begin{pmatrix} t_2 \\ j_2+1 \\ i_2+1 \\ t_2 \\ j_2+1 \\ i_2+1 \end{pmatrix}$$

Figure V.9: Generated Mappings for Two Statements of Jacobi

IV.1.2. Cholesky Factorization

The original source code and the transformed code generated by our algorithm for Cholesky factorization shown at Figure V.10.

```
    for k = 1, N
S1:     A(k,k) = sqrt(A(k,k))
        for i = k+1, N
S2:        A(i,k) = A(i,k) / A(k,k)
           for j = k+1, i
S3:           A(i,j) -= A(i,k) * A(j,k)
           end for
        end for
    end for
```

```
    for k = 1, N
        for i = 1, k-1
           for j = k, N
S1:           A(j,k) -= A(j,i) * A(k,i)
           end for
        end for
S2:     A(k,k) = sqrt(A(k,k))
        for i = k+1, N
S3:        A(i,k) = A(i,k) / A(k,k)
        end for
    end for
```

Figure V.10: (a) Original Code and (b) Transformed Code for Cholesky

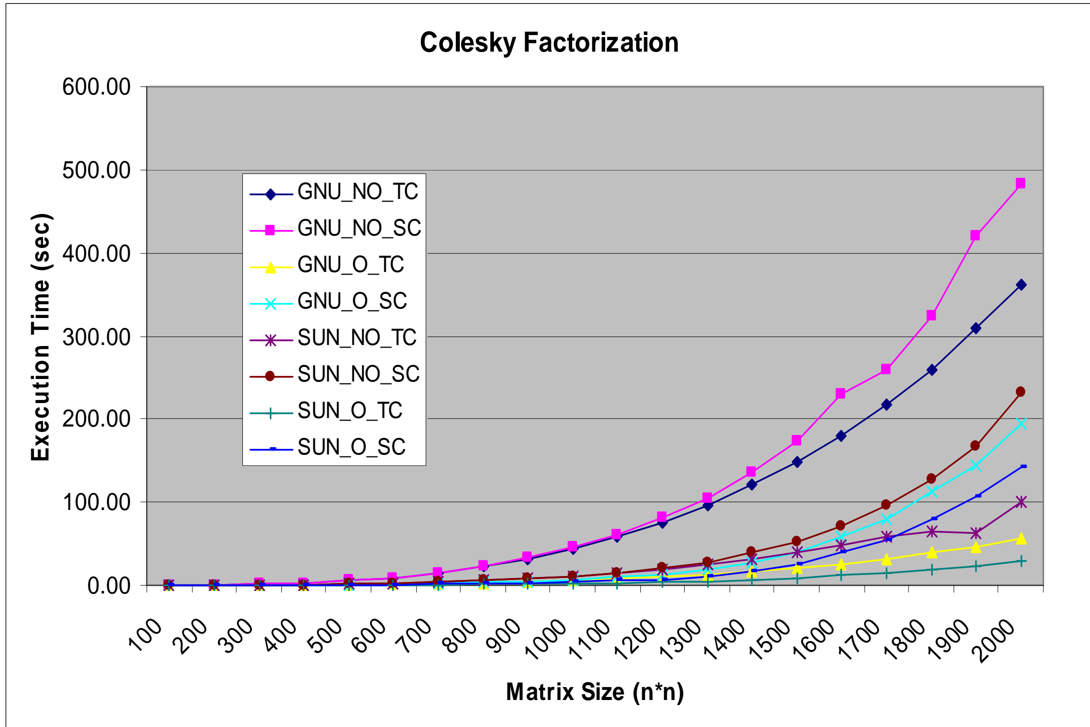


Figure V.11: Graphical Performances for Cholesky Factorization

Matrix Size(n*n)	GNU				SUN			
	Not Optimized		Optimized (-O3)		Not Optimized		Optimized (-xO5)	
	TC	SC	TC	SC	TC	SC	TC	SC
100	0.05	0.05	0.01	0.01	0.01	0.01	**0.00**	0.00
200	0.34	0.34	0.04	0.06	0.08	0.09	**0.01**	0.03
300	1.16	1.17	0.15	0.19	0.28	0.30	**0.05**	0.09
400	2.77	2.78	0.37	0.46	0.66	0.70	**0.14**	0.23
500	5.42	5.44	0.72	0.88	1.40	1.48	**0.28**	0.46
600	9.35	9.40	1.27	1.51	2.44	2.59	**0.48**	0.81
700	14.83	15.10	2.02	2.40	3.80	3.94	**0.79**	1.31
800	22.12	23.72	3.04	3.69	5.41	5.71	**1.15**	1.94
900	31.61	33.89	4.23	5.18	7.65	8.25	**1.64**	2.76
1000	43.99	46.82	5.86	7.08	10.48	11.13	**2.27**	3.83
1100	57.58	60.07	7.82	9.53	14.00	14.90	**3.06**	5.28
1200	75.17	82.45	10.19	12.90	18.28	19.89	**4.01**	7.31
1300	95.53	104.72	12.86	18.26	24.06	27.13	**5.20**	10.82
1400	121.07	136.46	16.56	28.03	30.84	39.13	**6.84**	16.36
1500	148.89	174.51	20.81	39.99	38.80	51.87	**9.23**	25.46
1600	180.35	229.57	25.98	58.45	47.88	71.12	**11.58**	39.14
1700	216.99	259.77	31.83	79.60	58.89	97.04	**14.91**	54.52
1800	260.06	323.77	39.62	111.95	64.23	128.23	**18.60**	78.68
1900	309.25	421.03	46.86	145.22	62.35	166.92	**23.04**	106.65
2000	361.27	483.46	57.29	194.72	99.89	231.84	**28.77**	141.55

Figure V.11: Numerical Performances for Cholesky Factorization

Figure V.11 shows the execution times. Again the best is transformed code with optimization by sun compiler. On the another point of view, if we compare the results of transformed code with no sun compiler optimization and source code with sun compiler optimization we conclude that in this case our framework makes better optimization than sun compiler optimizer. Figure V.12 shows chosen mappings by our framework for the Cholesky Factorization.

$$F_1\begin{pmatrix}k_1\end{pmatrix} = \begin{pmatrix}k_1\\k_1\\k_1\\k_1\\k_1\\k_1\end{pmatrix} \quad F_2\begin{pmatrix}k_2\\i_2\end{pmatrix} = \begin{pmatrix}k_2\\k_2\\i_2\\k_2\\i_2\\k_2\end{pmatrix} \quad F_3\begin{pmatrix}k_3\\i_3\\j_3\end{pmatrix} = \begin{pmatrix}j_3\\k_3\\i_3\\k_3\\i_3\\j_3\end{pmatrix}$$

Figure V.12: Generated Mappings for three statements of Cholesky Factorization

PART VI

CONCLUSIONS

Cache misses because of worse data locality are the most important factor on performance of the programs. The cache misses occurs at loop nests are attractive for the compiler commodity since there must be another way doing the same job with less cache misses in the nested loops. Especially image processing, video transmission, scientific programs etc spend a large percent of their time inside loops. In this study, we introduce a hybrid genetic algorithm for improving data locality of the programs with the objective of minimizing the execution time of the programs.

The experimental evaluation based on real application codes reveals that our hybrid genetic algorithm is generates good results as much as compiler optimizations and at some cases it outperforms the compiler optimizations. When we do not apply the compiler optimization our transformed code is always has a minimum execution time then the original code.

This thesis does not handle the data transformation techniques. Research is required further study this issue. The combination of data transformation and loop transformation can give a better performance. Also an additional research is required to calculate real cache misses count to be more accurate.

REFERENCES

[1]. N. Ahmed, N. Mateev, and K.Pingali. Synthesizing Transformations for Locality Enhancement of Imperfectly-nested Loops. ACM Intl. Conf. On Supercomputing, **2000**.

[2]. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In Workshop on Languages and Compilers for Parallel Computing (LCPC'03), LNCS, College Station, Texas, October **2003**.

[3]. Susan L.Graham David F.Bacon and Olivier J.Sharp. Compiler transformations for high-performance computing. Technical report, University of California, **1994**.

[4]. S. Chatterjee, E. Parker, P. Hanlon, A. Lebeck, Exact analysis of the cache behavior of nested loops, in: Proc. ACM SIGPLAN’01 Conference on Programming Language Design and Implementation (PLDI’01), **2001**, pp. 286–297.

[5]. W. Kelly and W. Pugh. “The Omega Calculator and Library”. Technical report, University of Maryland, November **1996**.

[6]. S. Coleman and K. McKinley, Tile Size Selection using Cache Organization and Data Layout. Proc. Programming Language Design and Implementation, ACM Press, **1995**.

[7]. T. Kisuki, P.M.W. Knijenburg, and M.F.P. O’Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In Proc. PACT, pages 237-246, **2000**.

[8]. P.M.W. Knijenburg, T. Kisuki, K. Galliyan and M.F.P. O’Boyle, The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling, Proc. FDDO-3, pages 31-40, **2000**.

[9]. K.S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems, 18(4) : 424-453, July **1996**.

[10]. A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing, Springer Verlag, **2003**.

[11]. T.Back, Evolutionary Algorithms in Theory and Practice, Oxford University Press, New York, **1996**.

[12]. D.E. Goldenberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley, **1989**.

[13]. Melanie Mitchell, An Introduction to Genetic Algorithm, MIT Press, **1998**.

[14]. P.A. Moscato, On evolution, search, optimization, genetic algorithms and material arts: Towards memetic algorithms, Tech. Rep. Caltech Concurrent Computation Program Report 826, Caltech, **1989**.

[15]. Gabriel Rivera, Chau-Wen Tseng, Data transformations for eliminating conflict misses, In SIGPLAN, 1998.

[16]. M. Kandemir. "Impact of Data Transformations on Memory Bank Locality," IEEE vol. 01, no. 1, p. 10506, Design, **2004**.

[17]. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loop nests. In Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), page 31, **2000**.

[18]. M.J. Wolfe. More iteration space tiling. Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pages 655-664, November **1989**.

[19]. G. Rivera and CW. Tseng. Data transformations for eliminating conflict misses. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 38-49, June **1998**.

[20]. M.D. Smith. Tracing with pixie. Technical report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November **1991**.

[21]. A. Schrijver. Theory of linear and integer programming. John Wiley & Sons, Inc., 1986

[22]. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In Proc. ACM SIGPLAN '91, pages 39-50, June **1991**.

[23]. P. Quinton, S. V. Rajopadhye, and T. Risset. On manipulating z-polyhedra using a canonical representation. Parallel Processing Letter, 7(2): 181-194, **1997**.

[24]. Z. Michalewicz and D.B. Fogel, How to Solve it: Modern Heuristics, Springer Verlag, 2000.

[25]. P. Moscato, Memetic Algorithms' Home Page www.densis.fee.unicamp.br/~moscato/memetic_home.html

[26]. P. Feautrier. Some efficient solutions to affine scheduling problem: one dimensional time. International Journal of Parallel Programming, 21(5):313-348, October **1992**.

[27]. G. Kreisel and J. L. Krevine. Elements of Mathematical Logic. North-Holland Pub. Co., **1967**.

[28]. J. J. Greffenstette, Incorporating Problem Specific Knowledge into a Genetic Algorithm, in Genetic Algorithms and Simulated Annealing, L.Davis (Ed.), Morgan Kaufmann Publishers, **1987**.

[29]. M. J. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, **2000**.

[30]. Grigori G. Fursin, Ph.D. thesis, Iterative Compilation and Performance Prediction for Numerical Applications, University of Edinburgh, January **2004**.

[31]. Web site for cache related info: www.nku.edu/~foxr/CSC462/NOTES/ch5.ppt

[32]. Web site for GNU compiler optimization http://developer.apple.com/documentation/DeveloperTools/gcc-3.3/gcc/Optimize-Options.html

[33]. Web site for SUN compiler optimization. http://developers.sun.com/prodtech/cc/articles/US3Cu/US3Cu.content.htm

APPENDIX A

FOURIER-MOTZKIN ELIMINATION

Fourier Motzkin elimination method eliminates a variable at a time in a system of linear inequalities. The remaining inequalities, which include some new ones, have a solution if, and only if, the original system has a solution. Eventually, this reduces to one variable or an inconsistency is determined during the elimination. One variable can be assigned any value in its range, and then a backtracking procedure can be executed to obtain values of the other variables, which were eliminated. Originally presented by Fourier in 1826, Motzkin analyzed it in 1933 in light of new developments of the theory of linear inequalities to solve linear programs.

The key idea is eliminating the variables one by one without changing the solution space. If the reduced system has a solution, the original system has a solution too. Let us consider a system of linear inequalities $Ax + b \geq 0$, where A is a matrix and b is a constant vector given at Figure A.1:

$$m \text{ constraints} \left\{ \underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots\cdots & a_{1n} \\ a_{21} & a_{22} & \cdots\cdots & a_{1n} \\ \cdots & \cdots & \cdots\cdots & \cdots \\ \cdots & \cdots & \cdots\cdots & \cdots \\ \cdots & \cdots & \cdots\cdots & \cdots \\ a_{m1} & a_{m2} & \cdots\cdots & a_{mn} \end{pmatrix}}_{n \text{ unknowns}} \right. \begin{bmatrix} x_1 \\ x_2 \\ .. \\ .. \\ .. \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ .. \\ .. \\ .. \\ b_n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ .. \\ .. \\ .. \\ 0 \end{bmatrix}$$

Figure A1: System of Linear Equalities with *m* Constraints and *n* Unknowns

The steps of algorithm can be described as below:

1. Choose an unknown x_i to eliminate.
2. Forms groups of inequalities as lower bounds that have positive coefficient of x_i, upper bounds that have negative coefficient of x_i and others that have not x_i. Arrange these groups as follows:

   Lower Bound Inequalities $\leq l\ x_i$

   $u\ x_i \leq$ Upper Bound Inequalities

   $0\ x_i \leq$ Other Inequalities

   where l and u is the coefficient of x_i.
3. Create new equalities by setting each inequality in lower bound group less or equal to each inequality in upper bound group. If we say the number of inequalities in the lower bound is *lb* and in upper bound is *ub* then at the end the total number of inequalities is equal to *lb***ub* + others. Arrangement of upper bound groups and lower bound groups to eliminate x_i can be done as follows:

   u * Lower Bound Inequalities $\leq l{*}u\ x_i \leq l$ * Upper Bound Inequalities

   At the end of elimination, the following inequalities are obtained:

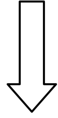
   u * Lower Bound Inequalities $\leq l$ * Upper Bound Inequalities

   $0\ x_i \leq$ Other Inequalities
4. Elimination of unknowns is repeated until a single unknown remains.

If we get a not empty space at the end, the original system has a solution. But the reverse is not true. Figure A.2 shows an example for two groups of affine inequalities.

Example 1: Example 2:

$$x + 2y - 2 \geq 0$$
$$x - 2y + 3 \geq 0$$
$$-x + 1 \geq 0$$



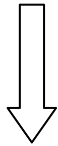
Decide to eliminate unknown x

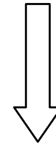
$$x + 2y - 2 \geq 0$$
$$x - 2y - 1 \geq 0$$
$$-x + 1 \geq 0$$

$$-2y + 2 \leq x$$
$$2y - 3 \leq x$$
$$x \leq 1$$

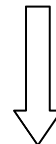
Lower bound and upper bound groups

$$-2y + 2 \leq x$$
$$2y + 1 \leq x$$
$$x \leq 1$$



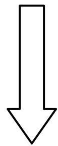


Rearrange the groups to eliminate x



$$-2y + 2 \leq 1$$
$$2y - 3 \leq 1$$

$$-2y + 2 \leq 1$$
$$2y + 1 \leq 1$$



Single unknown, y remains. So exhibit the range of y

$$1 \leq 2y$$
$$2y \leq 4$$

$$1/2 \leq y \leq 2$$

The range of y is not empty, so the original system inequalities has a solution.

$$1 \leq 2y$$
$$2y \leq 0$$

$$1/2 \leq y \leq 0$$

The range of y is empty, so the original system inequalities may or not have a solution.

Figure A2: Two Examples of Applying Fourier Motzkin Elimination.

APPENDIX B

FARKAS LEMMA

Farkas Lemma is a basic theory of the polyhedral model, which is proven by the Hungarian physicist and mathematician Gyula Farkas in 1901. Farkas Lemma comes in several alternatives forms. The affine form of Farkas Lemma is a particular for affine schedule purpose; it allows us to determine constraints from the dependencies.

Lemma 1: Farkas Lemma Affine Form [21]:

Let D be a nonempty polyhedron defined by the inequalities $Ax + b \geq 0$, then any affine function is nonnegative every where in D iff it is a positive affine combination:

$$f(x) = \lambda_0 + \lambda^T(A\,x + b), \text{ with } \lambda_0 \geq 0 \text{ and } \lambda^T \geq 0.$$

λ_0 and λ^T are called affine *Farkas multipliers*.

In other words, any function that is positive everywhere in a polyhedron $Ax + b \geq 0$ can be expressed as a positive linear combination of the rows of the vector $Ax + b$. As an example, let us consider an affine function $f(x) = ax + b$. We want to know the relation between b and a, so that the function f(x) can be non-negative everywhere in the domain $1 \leq x \leq 5$. It easy to see geometrically that if a is positive then $b \geq a$, and if a is negative then $b \geq -5$. Farkas Lemma can be used to deduce these constraints algebraically. Figure B.1 shows Farkas Lemma application to this example.

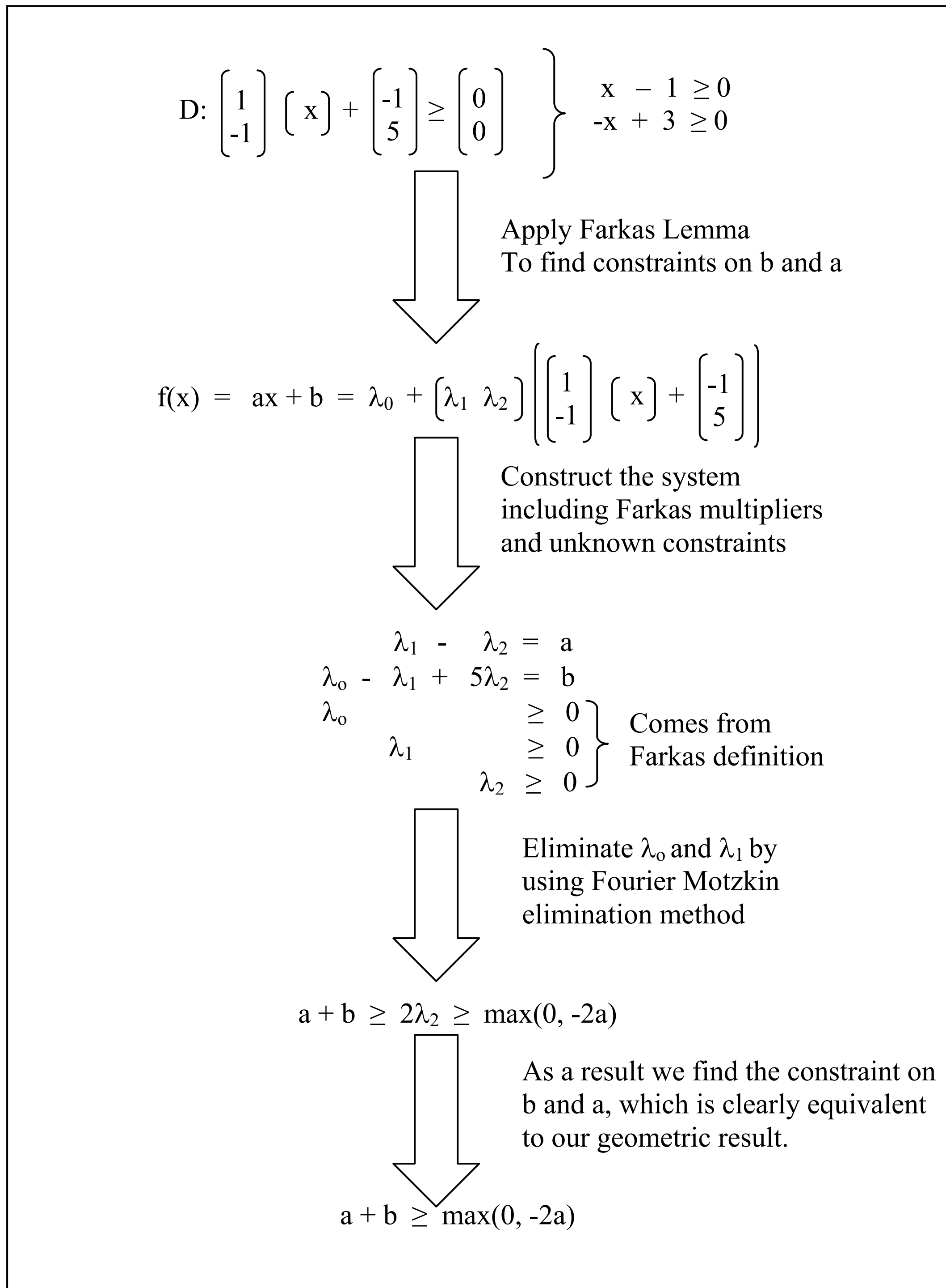


Figure B.1: An Example to Implementation of Farkas Lemma.

Related to our study:

We use Farkas Lemma, on our dependencies. We want to obtain difference vectors of the mappings is always greater than or equal to zero on the domain of corresponding dependence. For instance, Let apply the Farkas Lemma to the dependence domain (see section IV1.3.) given below, and gets the constraint.

$D_1 = \{ [i_1, j_1, k_1, i_2, j_2] : 1 \leq i_1, i_2 \leq M, 1 \leq j_1, j_2 \leq N, 1 \leq k_1 \leq j_1\text{-}1, j_1 = j_2, i_1 = i_2 \}$

$$f(x) = \lambda_o + \lambda_1(i_1\text{-}1) + \lambda_2(M\text{-} i_1) + \lambda_3(i_2\text{-}1) + \lambda_4(M\text{-} i_2) + \lambda_5(j_1\text{-}1) + \lambda_6(N\text{-} j_1)$$
$$+ \lambda_7(j_2\text{-}1) + \lambda_8(N\text{-} j_2) + \lambda_9(k_1\text{-}1) + \lambda_{10}(j_1\text{-}1 - k_1) + \lambda_{11}(j_1\text{-} j_2)$$
$$+ \lambda_{12}(j_2\text{-} j_1) + \lambda_{13}(i_1\text{-} i_2) + \lambda_{14}(i_2\text{-} i_1)$$

We want to make difference vector entries grater or equal to zero, f(x) comes from the difference vector of mappings of the dependent statements.

$$F(x) = a\, i_1 + b\, i_2 + c\, k_1 + d\, j_1 + e\, j_2 + f\, M + g\, N + h$$

Coefficient of i_1 = a = $\lambda_1 - \lambda_2 + \lambda_{13} - \lambda_{14}$

Coefficient of i_2 = b = $\lambda_3 - \lambda_4 - \lambda_{13} + \lambda_{14}$

Coefficient of j_1 = c = $\lambda_5 - \lambda_6 + \lambda_{10} + \lambda_{11} - \lambda_{12}$

Coefficient of j_2 = d = $\lambda_7 - \lambda_8 - \lambda_{11} + \lambda_{12}$

Coefficient of k_1 = e = $\lambda_9 - \lambda_{10} + \lambda_{11} - \lambda_{12}$

Coefficient of M = f = $\lambda_2 + \lambda_4$

Coefficient of N = g = $\lambda_6 + \lambda_8$

Coefficient of constant term = h = $\lambda_0 - \lambda_1 - \lambda_3 - \lambda_7 - \lambda_9 - \lambda_{10}$

After eliminating unknown Farkas multipliers, we get the following constraint on unknown coefficients of difference vector f(x). As a result we can check the legality of our mappings, by using these constraints from Farkas Lemma. We use omega calculator Farkas method to get these constraints.

$$H + 2f + 2a + 2c + e + 2b + 2d \geq 0$$
$$f + c + e + d \geq 0$$
$$f + c + d \geq 0$$
$$f + a + c + b + d \geq 0$$
$$f + a + c + e + b + d \geq 0$$
$$f + a + b \geq 0$$
$$h + 2f + a + 2c + e + b + 2d \geq 0$$
$$f \geq 0$$

APPENDIX C

OMEGA CALCULATOR AND PROJECT

The goal of the Omega project is constructing frameworks for the analysis and transformation of scientific programs. The software includes:

- The Omega library, a set of routines for manipulating linear constraints over integer variables, integer tuple relations and sets.
- The code generation library, a set of routines for generating code to scan the points in the union of a number of convex sets. (requires the Omega library)
- The Omega calculator, a text-based interface to the Omega library (requires both above libraries)
- The Uniform library, a source to source parallelizing transformation system,
- Petit, a educational/research tool for analyzing array data dependences (needs all three libraries)

Omega library includes a C++ class that manipulates integer tuple relations and sets, such as:

$\{[i,j] \rightarrow [j,j'] : 1 \leq i < j < j' \leq N\}$ AND $\{[i,j] : 1 \leq i < j \leq N \}$

Relations and sets are described using Presburger formulas [27] a class of logical formulas which can be built from affine constraints over integer variables, the logical connectives $\neg$, $\wedge$ and $\vee$, and the quantifiers $\forall$ and $\exists$. In the literature there are many research, which have used the omega library.

Omega calculator is a text-based interface to omega library. The usage of omega calculator is easier then library. We give an example at Figure C.1 to compare the usage of them. In this example we try to construct a relation by using omega calculator and library.

```
{
  [i1,j1,k1] -> [i2,j2,k2] : i2 = i1 && j1 < j2 || i2 = i1 && j2 = j1 && k1 <= k2 || i1 < i2
}
```

```
char *_1[3] = {"i1", "j1", "k1"};
char *_2[3] = {"i2", "j2", "k2"};
Variable_ID id1[3];
Variable_ID id2[3];
Variable_ID &i1=id1[0], &j1=id1[1], &k1=id1[2];
Variable_ID &i2=id2[0], &j2=id2[1], &k2=id2[2];

Relation s(3,3);
for (int i=0; i<3; i++)
{
    s.name_input_var(i+1,_1[i]);
    s.name_output_var(i+1,_2[i]);
    id1[i]=s.input_var(i+1);
    id2[i]=s.output_var(i+1);
}

F_And *a1 = s.add_and();
F_Or *o1 = a1->add_or();
F_And *a2 = o1->add_and();
GEQ_Handle h1 = a2->add_GEQ();
h1.update_coef(i1,1);
h1.update_coef(i2,-1);
h1.negate();
F_And *a3 = o1->add_and();
EQ_Handle h2 = a3->add_EQ();
h2.update_coef(i1,1);
h2.update_coef(i2,-1);
F_Or *o2 = a3->add_or();
F_And *a4 = o2->add_and();
GEQ_Handle h3 = a4->add_GEQ();
h3.update_coef(j1,1);
h3.update_coef(j2,-1);
h3.negate();
F_And *a5 = o2->add_and();
EQ_Handle h4 = a5->add_EQ();
h4.update_coef(j1,1);
h4.update_coef(j2,-1);
GEQ_Handle h5 = a5->add_GEQ();
h5.update_coef(k1,-1);
h5.update_coef(k2,1);
```

Figure C.1: Relation Definition by Using (a) Omega Calculator, (b) Omega Library

OC read its input from a file as its first argument and writes its output to the standard output. Usage of OC requires write to a file and read from file operations. Also the input file must be written in the format of OC and then output file must be parsed. Figure B.2 gives some input and output examples of OC.

```
# R := {[i] -> [i’] : 1 <= i, i’ <= 10 && i’ = i + 1};
# R;
{ [i] -> [i+1] : 1<= i <=9 }

# inverse R;
{ [i] -> [i-1] : 2 <= i <= 10 }

# range R;
{ [i] : 2 <= i <= 10 }

# S := { [i] : 5 <= i <= 25 }
# R\S;
{ [i] -> [i+1] : 5 <= i <= 9  }
…………………………..
```

Figure C.2: Some example to the OC methods.

Input consists some statements terminated by semicolons. The “#” character at the beginning of line indicates that the rest of line is comment. Also the output includes the lines which starts with the character “#”. The character “#” in the output file indicates the input line to the omega calculator in other words the processed lines. In the OC relations and sets are defined by using presburger formula operators (And, Or, Not, Exists, …etc.) and comparison operators (>=, <=, <, > , =, !=). Also relational operations (Union, Intersection, Convex Hull, Farkas, … etc.) can be applied on relations and sets. The list of all operations can be found in [5].

In our problem we show dependencies as sets and mappings as relations. We applied the “Farkas” method on the dependence relations to define our constraints for generating valid solutions. We used the code generation facility of the OC to generate code of the best solution. Our iteration spaces and mappings are described as sets and relations respectively. The best solution statement mappings and iteration spaces are given to the OC as input to generate transformed code.

GÜLŞAH YILMAZ

Fulya Mah. Fulya Bayırı Sk. No: 35 D: 4 Taç Apt. Şişli/ İSTANBUL

E-mail: gulsahyilmaz20@yahoo.com

Phone: +90 532 709 04 73 / +90 212 272 81 60

CAREER OBJECTIVES

To acquire a career in information technologies by using my abilities, knowledge and sense technology.

EDUCATION

2002-Present Marmara University (MS in Computer Engineering), ISTANBUL
1997-2002 Marmara University (BS in Computer Engineering), ISTANBUL
1993-1996 Cumhuriyet High School, ESKİŞEHİR
1990-1993 Atatürk Secondry School, ESKİŞEHİR
1985-1990 Ülkü Primary School, ESKİŞEHİR

WORK EXPERIENCE

2002-Present Sesam Yazılım as Project Leader

PROFESSIONAL SKILLS

English (fluent)

RESEARCH INTERESTS

Optimization Problems
Genetic Algorithms

PUBLICATIONS

- H. Topçuoğlu, F. Corut, M. Ermis, G. Yılmaz: Solving the uncapacitated hub location problem using genetic algorithms, Computers & Operations Research 32 (2005) 967-984

PERSONAL

Born on August 30, 1979 in ANKARA
Reading, Music, Cinema, Sport

REFERENCES

Available upon request