國立交通大學

資訊科學與工程研究所

碩士論文

藉由多面體模型 (Polyhedral Model) 改善同質多核心系統之資料局部性

Data locality improvement by Polyhedral Model for Homogeneous Multiprocessors

研 究 生：高淑娟

指導教授：單智君　博士

中 華 民 國　九 十 九 年 七 月

藉由多面體模型 (Polyhedral Model) 改善同質多核心系統之資料局部性

Data locality improvement by Polyhedral Model for Homogeneous Multiprocessors

研 究 生：高淑娟　　Student：Shu-Chuan Kao

指導教授：單智君　　Advisor：Dr. Jyh-Jiun Shann

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Engineering

July 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年七月

藉由多面體模型 (Polyhedral Model) 改善同質多核心系統之資料局部性

學生：高淑娟　　指導教授：單智君　博士

國立交通大學資訊科學與工程研究所碩士班

摘要

在現今同質多核心(homogeneous multiprocessors)系統上，常用的平行編程模式(parallel programming model)有 OpenMP 與 MPI 等，其中又以 OpenMP 最為被廣泛地使用。另一方面，對於同質多核心系統而言，效能的瓶頸通常是在迴圈程式的記憶體存取上。因此，為提昇程式在同質多核心系統上的執行效能，程式開發者會藉由一些迴圈優化技術來提高程式執行的資料局部性(data locality)以減少外部記憶體(external memory)的存取次數，其中常見的技術有迴圈互換(loop interchange)與迴圈區塊化(loop tiling)等。為了執行這些優化的技術，迴圈程式碼都通常會先轉成抽象化中間表示式(intermediate representation, IR)；在執行完一連串優化技術後，再將 IR 轉回成程式碼或可執行檔。目前常見的 IR 表示方式有抽象語法樹 (abstract syntax tree, AST)與多面體模型(polyhedral model)等。對於迴圈程式碼而言，使用 polyhedral model 可以減少避免轉換順序之間造成的副作用(side-effect)，如程式碼變大及複雜度變高等問題。

在本論文中，我們針對迴圈程式的資料局部性的優化實作了一套迴圈程式原始碼到

原始碼(source-to-source)的轉換框架(framework)。本論文開發之 framework 可自動找出 OpenMP 程式中的 DOALL 迴圈，並針對這些迴圈進行 loop tiling 以提昇資料局部性。根據模擬結果顯示，透過所開發之 loop tiling 技術，程式的執行效能平均可提昇 14.1%。此外，藉由本論文開發之 framework，程式開發者可以快速地發展在 polyhedral model 下所需之迴圈優化技術。

Data locality improvement by Polyhedral Model

for Homogeneous Multiprocessors

Student：Shu-Chuan Kao Advisor：Dr. Jyh-Jiun Shann

Institute of Computer Science and Engineering

National Chiao Tung University

Abstract

The parallel programming model on homogeneous multiprocessors includes OpenMP, MPI, and others. OpenMP is the most widely used in these common parallel programming models. On the other hand, the performance bottleneck on homogeneous multiprocessors is usually the memory access within a nested loop. Therefore, programmers would adopt some loop optimizations to enhance data locality and reduce external memory access times for the parallel execution on homogeneous multiprocessors. The common techniques of loop optimizations are loop interchange and loop tiling. The source code of the nested loops would usually translate to the intermediate representation (IR) for applying the loop optimizations. After a series of optimizations, IR would translate back to the source code or the executable program. The common IR includes abstract syntax tree, polyhedral model, and others. Using polyhedral IR could avoid causing side-effects between transformation orders, such as increasing the code size and enhancing the transformation complexity.

In this thesis, we present the implementation of a polyhedral source-to-source

transformation framework that could optimize the nested loops for the data locality improvement. The framework could find the DOALL loops automatically and adopt loop tiling to enhance the data locality. According to the simulation results, the programs could increase 14.1% performance in average by performing loop tiling. Moreover, programmers could develop loop optimizations quickly based on our polyhedral framework.

致謝或序言

首先要感謝我的指導老師 單智君教授，在這兩年對於學生的細心指導與建議，且不時的督促及勉勵學生，並使我學習到如何發掘問題，以及如何克服問題，進而培養獨立研究的能力。而在 Group meeting 上， 楊武教授、 徐慰中教授及 游逸平教授都給予學生許多寶貴的意見與建議，讓學生可以多方思考與改進，對於這些曾教導過我的教授們，學生真的是非常感激。

感謝奕緯學長及裕生學長帶領我進入 Compiler 相關的領域，不僅僅只是給予我研究上的建議，還時常抽空與我進行討論，平日閒暇之餘也常鼓勵我或提供我養生的秘訣，最終才使得我學習到相關知識並完成此研究。同時，實驗室的學長姐、同儕以及學弟妹，在這一起渡過的二年時光中，感謝你們不僅僅是我研究路途上的好夥伴，更在我遇到低潮時給予幫助，我會永遠感念在心。

最後，對於我的家人以及總是給予鼓勵並陪伴著我的親友們，淑娟也在此獻上最誠摯的謝意。

高淑娟 2010.8.5

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Research Motivation

Homogeneous multi-processors are popular recently and the parallel programming models such as OpenMP are widely used. The OpenMP is an important method and language extension for program parallelism. In order to make existing OpenMP applications to achieve higher performance, we want to improve data locality in nested loops. Moreover, we not only optimize the data movement and allocation, but also decrease cache misses in memory hierarchy.

In addition, the growing speed gap between memory and processor makes an efficient use of the cache more important to get higher performance. Tiling is a critical optimizing transformation for data locality improvement. It groups points in an iteration space into smaller tiles (blocks) allowing reuse when the block in a faster memory hierarchy (e.g. registers or cache).

Because applying syntactic transformations such as abstract syntax tree would dramatically increase the code size and transformation complexity, this work adopts another powerful representation, called “Polyhedral model”. In the polyhedral model, the program is viewed as a “statement centric” structure; therefore many side-effects could be avoided.

In this thesis, we proposed a source-to-source transformation framework in which realizes polyhedral model and loop tiling to improve the data locality of nested loops.

1.1.1 Limitation of Syntactic Transformations

Current compilers provide an unstructured search space for syntactic transformations, i.e., control structures are regenerated after each transformation and it's unable to perform a serial of transformations well. There exist some limitations in syntactic intermediate representations (IR).

Code Size and Transformation Complexity – The code size and transformation complexity would dramatically increase after several transformations. Consider the simple synthetic example shown in Figure 1.1 and the analysis depicted in Figure 1.2.

```
For (i=0; i<M; i++)
        Z[i] = 0;
        For (j=0; j<N; j++)
        Z[i] += (A[i][j] + B[j][i]*X[j]);
For (k=0; k<P; k++)
        For (l=0; l<Q; l++)
                Z[k] += A[k][l]*Y[l]);
```

Figure 1.1 Source code for code size and complexity analysis

	Syntacitc (# lines)	Polyhedral (# values)
Original code	7	78
Outer loop Fusion	28 (x4.0)	78 (x1.0)
Inner loop Fusion	84 (x12.0)	78 (x1.0)
Fission	78 (x11.2)	78 (x1.0)
Strip-mine	223 (x31.8)	122 (x1.5)
Strip-mine	259 (x37.0)	182 (x2.3)
Interchange	290 (x41.4)	182 (x2.3)

Figure 1.2 Code size and complexity analysis of Figure 1.1

When applying some transformations in the source code in Figure 1.1, the code size would increase a lot in the syntactic representation. However, when applying same transformations in the polyhedral representation, only the strip-mining transformation increases the overall complexity. Because the polyhedral representation consists in a fixed number of matrices associated with each statement, only the dimension of some matrices slightly increases. Moreover, neither its transformation complexity increases nor its code size varies significantly.

Patterns Breaking -- When compilers look for the transformations opportunities depending on pattern-matching rules, it's difficult to find the suitable transformation pattern. As a consequence, prior transformations usually break target patterns for further ones. Hence, combining different loop transformations would not always lead to the best result.

Limitations Of Phase Ordering – the phase order of loop transformations is another challenge to achieve optimal result. Because every transformation would change the intermediate representation, the opportunity of the following transformations would decrease.

For these limitations[1], we believe that the polyhedral representation would be an appropriate way to avoid these side-effects.

1.1.2 Advantage and disadvantage of Polyhedral Model

This polyhedral representation could avoid side-effects addressed above. For example, it would not increase code size or/and transformation complexity significantly. Unfortunately, this polyhedral representation is not easy to understand for programmers.

1.2 Research Objective

Many compute-intensive applications often spend most of their execution time in nested loops. The polyhedral model provides a powerful abstraction on loop representation in which each statement in an iteration is viewed as an integer point in a well-defined space called the statement's *polyhedron*. With representation for each statement, it is possible to show the correctness of transformations in a completely mathematical setting relying on machinery from linear algebra and integer linear programming.

The task of program optimization for data locality in the polyhedral model may be viewed in terms of three phases: (1) constructing the loop information from intermediate representation of the abstract syntax tree to the polyhedral model (2) performing transformations for data locality improvement in the polyhedral intermediate representation, and (3) generating the target code from the transformed intermediate

representation of the polyhedral model.

1.3 Organization of this Thesis

The rest of this paper is organized as follows: Section 2 provides the mathematical background and the data structure for the polyhedral model. Chapter 3 describes the detail of our framework. Chapter 4 presents the experimental results and discussion follows. Finally, Section 5 dedicates to the conclusions and the future work planed.

Chapter 2 Background

In this chapter, we give an overview of the polyhedral model, and introduce notations used throughout the dissertation. In syntactical compilers, two important structuring elements are the basic block (BB) and the control flow graph (CFG). However, the most important element in the data structure used in the polyhedral model is the statements (S) in nested loops.

2.1 Polyhedral Model

The polyhedral model[2-3] is applicable to the nested loops which the data access and loop bounds are affine functions (means linear function with a constant). Accordingly, we could adopt the program transformations more efficiently.

2.1.1 Polyhedron

Definition 1 - Polyhedron (a.k.a. Polyhedral). A *polyhedron* is an intersection of a finite number of half-spaces. A polyhedron also has a representation in terms of vertices, rays, and lines, and algorithms like the Chernikova algorithm[4].

Here is an example for the polyhedron. The nested loops in source code may like Figure 2.1, and the corresponding polyhedron is in Figure 2.2.

```
for (i=2; i<N; i++)
    for (j=2; j<N; j++)
        A[i] = pi;
```

Figure 2.1 Source code for polyhedron representation

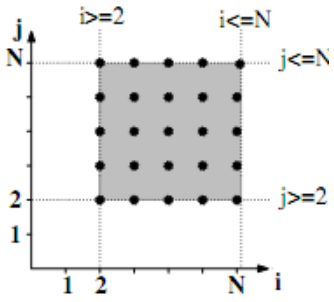


Figure 2.2 Corresponding polyhedron of Figure 2.1

Polylib[5] and PPL[6] are two of the libraries[7-8] that provide a range of functions to perform various operations on polyhedral. In this work, we are always interested in the integer points inside a polyhedron since the loop's iterators typically have integer data types and traverse an integer space.

2.1.2 Iteration Domain

The iteration domain represents the nested loops bound constraints, including upper bounds and lower bounds of an iterator. The iteration domain of a statement (S) is defined by:

$$D^S = \left\{ \vec{x} \mid \vec{x} \in Z^n, A\vec{x} \geq \vec{c} \right\}$$

We recall the previous simple example in Figure 1.1.

```
        For (i=0; i<M; i++)
S1          Z[i] = 0;
            For (j=0; j<N; j++)
S2          Z[i] += (A[i][j] + B[j][i]*X[j]);
        For (k=0; k<P; k++)
            For (l=0; l<Q; l++)
S3              Z[k] += A[k][l]*Y[l];
```

S1 (Statement one) belongs to one-dimensional iteration domain, meanwhile, S_2 and S_3 belong to two-dimensional iteration domains. The global parameters in

this nested loop are M, N, P and Q. As a result, the matrix of iteration domain has one column for each iterator and each global parameter, here respectively i, j and M, N, P, Q. Otherwise, we also keep one column for storing a constant value.

Moreover, the loop's iteration domain is bounded by affine inequalities which we called it loop bounds. Each row of the matrix presents an affine inequality. It means that for each iterator at least has an upper bound and a lower bound. Therefore the iteration domain in S_1 has two rows and six columns.

Hence, the iteration domain of S_1 is $D^{S_1} = \begin{array}{c} \begin{matrix} i & M & N & P & Q & 1 \end{matrix} \\ \left[\begin{array}{c|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & -1 \end{array}\right] \end{array}$

The first row presents $1\times i+0\times M+0\times N+0\times P+0\times Q+0\times 1\geq 0$, which means the lower bound in loop i is $i \geq 0$. Moreover, the second row presents $-1\times i+1\times M+0\times N+0\times P+0\times Q+0\times -1\geq 0$, which means the upper bound in loop j is $i \leq M-1$. Accordingly, the iteration domain of S_2 and S_3 is $D^{S_2} = \begin{array}{c} \begin{matrix} i & j & M & N & P & Q & 1 \end{matrix} \\ \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & -1 \end{array}\right] \end{array}$ and $D^{S_3} = \begin{array}{c} \begin{matrix} i & j & M & N & P & Q & 1 \end{matrix} \\ \left[\begin{array}{cc|cccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & -1 \end{array}\right] \end{array}$

The first two rows of the matrix are loop i's constrains, and the others are loop j's.

2.1.3 Schedule matrix

The schedule matrix characterizes the execution order of each statement within the nested loops. To achieve that, we define a timestamp to record execution

sequence of each statement. The schedule matrix θ^S of a statement is defined by

$$\Theta^S = \left[\begin{array}{ccc|c} 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d}^S & 0 \\ 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d}^S & 0 \\ \vdots & \ddots & \vdots & \vdots \\ A_{3,1}^S & \cdots & A_{3,d}^S & 0 \\ 0 & \cdots & 0 & \beta_{d^S}^S \end{array}\right]$$

Let d^S be the depth of the nested loops, which means how many levels from the outermost loop to the innermost loop. We call the outermost level of the nested loops depth 0. And A^S is the iteration vector, which means this statement surrounding by which iterators.

Let β^S be the static statement ordering vector, which represents the timestamp. It points out the execution ordering of this statement. The first execution part would be label as 0, and the second execution part would be label as 1, and so on. Next, we trace this nested loop recursively, and label each timestamp to all statements and loops. When the whole nested loop was traversed, we could specify the execution order for all statements in this nested loop.

So we recall the previous simple example again in Figure 1.1.

```
S1    For (i=0; i<M; i++)
          Z[i] = 0;
S2        For (j=0; j<N; j++)
          Z[i] += (A[i][j] + B[j][i]*X[j]);
      For (k=0; k<P; k++)
S3        For (l=0; l<Q; l++)
              Z[k] += A[k][l]*Y[l]);
```

The schedule matrix of S1 is $\theta^{S_1} = \left[\begin{array}{c|c} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right]$. The left-side of the matrix means that

this statement surrounding by loop i, and the β^S in the right-hand side of the matrix is [0, 0]. It means the stamement1 in depth 0 is the first part to be executed, and it's also the first part to be executed in depth 1.

Accordingly, the schedule matrix of S2 and S3 is

$$\theta^{S_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \qquad \theta^{S_3} = \left[\begin{array}{cc|c} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right]$$

We could translate the source polyhedron into the target polyhedron containing the same points but in a new coordinate system with a new execution order. As a result, we could distribute the iterations in space, i.e. across different processors, order them in time, or both. In some specify transformations, we would modify this matrix to adopt loop transformations.

2.1.4 Access Function

The array reference represents all array accesses in left-hand side or right-hand side of each statement. In other words, we store left-hand side or right-hand side array information of each statement in matrixes.

For each statement S, we define a pair of sets $\mathscr{L}_{hs}^{S}$ and $\mathscr{R}_{hs}^{S}$. Each pair represents an array reference in the left-hand side or right-hand side of the statement. The access function f is defined by a matrix F such that

$$f(i) = F \times [i, g, 1]^t$$

f is a function of loop iterators (i), local variables and global parameters (g), and a constant value (1). So we recall the previous simple example in Figure 1.1.

```
        For (i=0; i<M; i++)
S1          Z[i] = 0;
            For (j=0; j<N; j++)
S2          Z[i] += (A[i][j] + B[j][i]*X[j]);
        For (k=0; k<P; k++)
            For (l=0; l<Q; l++)
S3              Z[k] += A[k][l]*Y[l]);
```

Because the right-hand side array access of S1 is Z[i], we store the access function as $\mathscr{L}_{hs}^{S1} = \{(Z, [1 \mid 0\,0\,0\,0 \mid 0]^{\substack{j \\ 1}}_{M\,N\,P\,Q})\}$, which means that the S_1 refers to the array Z and after the comma is the parameter part. In addition, the left-hand side array access of S1 is the empty set, we store the access function as $\mathscr{R}_{hs}^{S1} = \{\ \}$

Accordingly, we could store the information for S2 and S3 as below,

$$\mathscr{L}_{hs}^{S2} = \{(Z, [10 \mid 0000 \mid 0]),$$

$$\mathscr{R}_{hs}^{S2} = \{(Z, [10 \mid 0000 \mid 0]),\ (A, \begin{bmatrix} 10 \mid 0000 \mid 0 \\ 01 \mid 0000 \mid 0 \end{bmatrix}),$$

$$(B, \begin{bmatrix} 01 \mid 0000 \mid 0 \\ 10 \mid 0000 \mid 0 \end{bmatrix}), (X, [01 \mid 0000 \mid 0])\}$$

$$\mathscr{L}_{hs}^{S3} = \{(Z, [10 \mid 0000 \mid 0])\}$$

$$\mathscr{R}_{hs}^{S3} = \{(Z, [10 \mid 0000 \mid 0]), (A, \begin{bmatrix} 10 \mid 0000 \mid 0 \\ 01 \mid 0000 \mid 0 \end{bmatrix}), (X, [01 \mid 0000 \mid 0])\}$$

These four equations mean the left-hand side and right-hand side array access's names and parameters used in each statement.

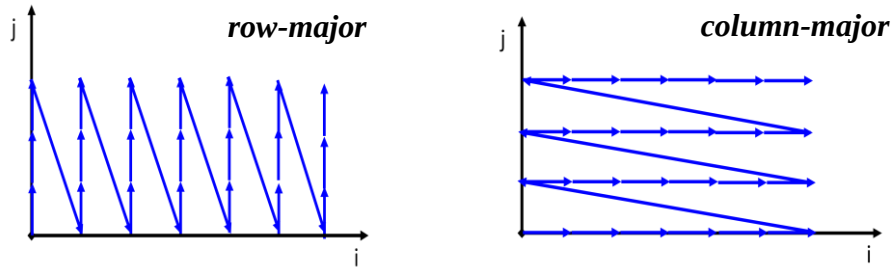
2.2 Optimization for Data Locality

The growing speed gap between memory and processor makes an efficient use of

the cache more important to reach high performance. One of the most important ways to improve cache behavior is to increase the data locality, such as making lower cache miss rates. Data locality could be improved by reordering the memory accesses so that the same array elements are accessed closer together.

Data use and reuse often occur in different iterations of the same loop. Two traditional transformations that improve the data locality in a single iteration are loop tiling and loop interchange.

Loop interchange, as known as loop coalescing, is to permute the order of the loops to modify the memory traversal order. If the array elements from the same row are stored consecutively (for example : a[1,1], a[1,2], a[1,3]...), namely row-major. If we access array elements from the same column together (for example : a[1,1], a[2,1], a[3,1]...) , called column-major.



(a) (b)

Figure 2.3 Different memory traversal orders. (a) row-major (b) column-major

Therefore, if the storage way is different with the access way, we could exchange the order of two iterators of the loops.

Loop tiling is applied when the long-distance reuses occur between different

iterations of a single outer loop. The principle idea is to process less datum in one iteration of the loop, so that datum could be retained in the cache between several iterations of the loop.

We partition the iteration space into tiles that may be executed concurrently on different processors. In other words, grouping points in an iteration space into smaller tiles allows reuse datum when the tile fits in a faster level of the memory hierarchy, such as the cache. The following Figure 2.4 is an example to divide the matrix into sub-matrices, or tiles.

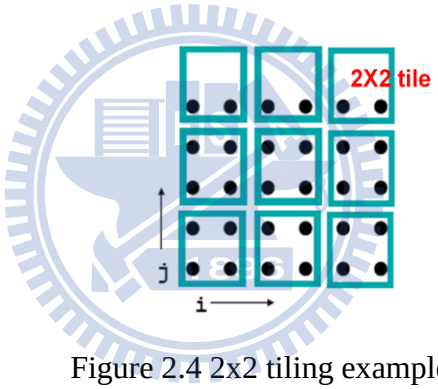


Figure 2.4 2x2 tiling example

Chapter 3 Implementation of Data Locality Transformation and Code Generation Algorithm

3.1 Overview

In this thesis, our goal is to modify a compiler framework for data locality improvement of OpenMP source code using polyhedral model. The flowchart of the framework may like Figure 3.1.

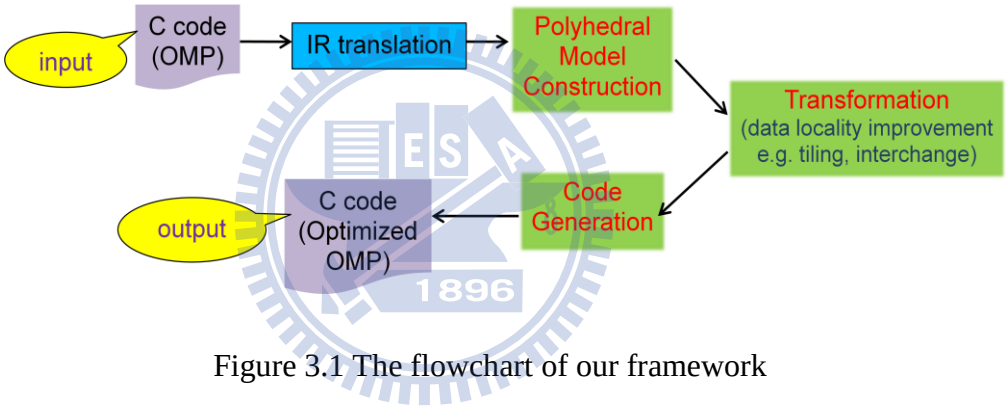


Figure 3.1 The flowchart of our framework

In the first part of this chapter, we would introduce how to translate from the abstract syntax tree of CETUS[9] to the matrices of polyhedral model. In the second part, the nested loops transformations for data locality would be introduced. Also, array accesses analysis in statements would be mention. In the last part of this chapter, we would introduce how to translate the matrices of polyhedral model back to abstract syntax tree of CETUS.

3.2 AST to Polyhedral Model

In this section, we want to translate the intermediate representations from the abstract syntax tree of CETUS to the matrices of polyhedral model. The input is an OpenMP source program. The output is the data structure in polyhedral model.

For CETUS compiler supporting, we could get the abstract syntax tree from the OpenMP source code. Accordingly, we need to extract loop information from this abstract syntax tree and keep the necessary information in polyhedral model's matrices, which were described in previous section.

First, we recognize all nested loops with OpenMP annotation in source program. For each nested loops, we use DepthFirstIterator in CETUS compiler to trace from the outermost level to the innermost level. Then we could get the information from each level of the nested loops recursively, and store them to the data structure ($\mathcal{D}$, $\mathcal{S}$, $\mathcal{A}$) in polyhedral model for each statement.

To achieve that, we propose an algorithm for scanning the nested loops in Figure 3.2.

Input : a nested loop ($\mathcal{L}$) with OpenMP annotation
Output : data structure ($\mathcal{D}$, $\mathcal{S}$, $\mathcal{A}$) in Polyhedral Model
1. Trace each level of $\mathcal{L}$, store the order to $\mathcal{S}$
2. If the node is loop, store the iterator and bounds to $\mathcal{D}$
3. If the node is statement, store the array access to $\mathcal{A}$
4. For each $\mathcal{L}\longrightarrow$ inside, apply step 1 to 3 to inside

Figure 3.2 The algorithm for scanning the nested loops

It means that when tracing the nested loops in abstract syntax tree from the outermost level to the innermost level, we should record the execution order at the same time. In addition, if this node is a loop form, we could use the method of getInitialStatement()、getCondition() and getStep() to get the information of the loop. For example, index variables, lower bounds, and upper bounds are all information which we need to store. Otherwise, if this node is a statement form, we could use the method of getBody() to record all array accesses in this statement. Then we trace to the inner level recursively until all nodes in this nested loops were traced.

Here is an example in Figure 3.3 to explain how it works. We could follow the blue arrowhead to trace all nodes in the loop. At first, we encounter a for-form node and record the information of the iteration domain and the schedule matrix.

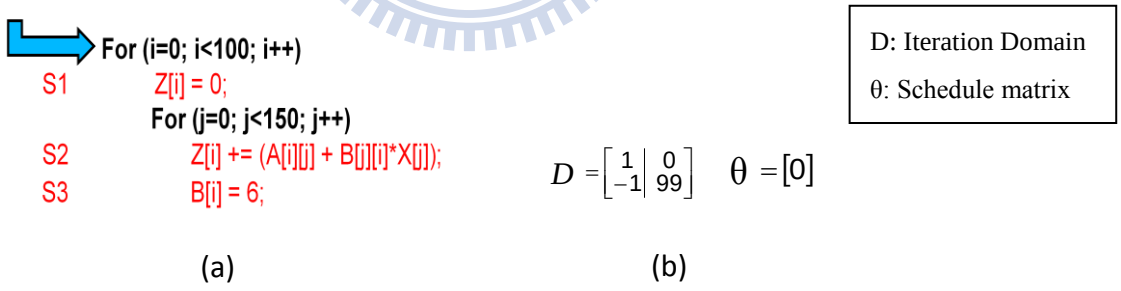


Figure 3.3 (a) Source code for explaining the polyhedral construction (b) temporary result

Then we trace the inner level of the loop, and encounter a statement-form node. So we modify the schedule matrix and store all temporary information to the S_1, as shown in Figure 3.4.

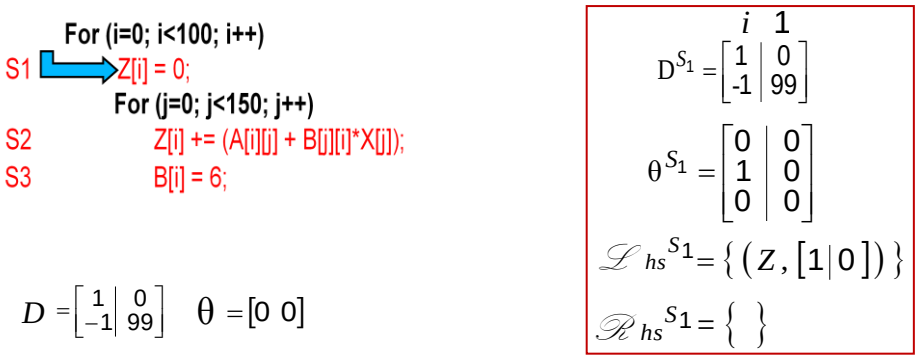
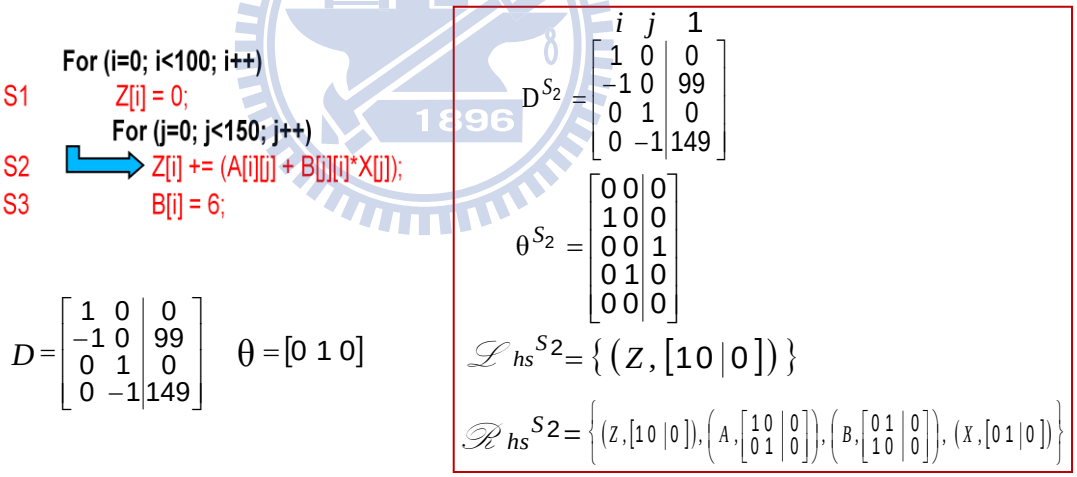
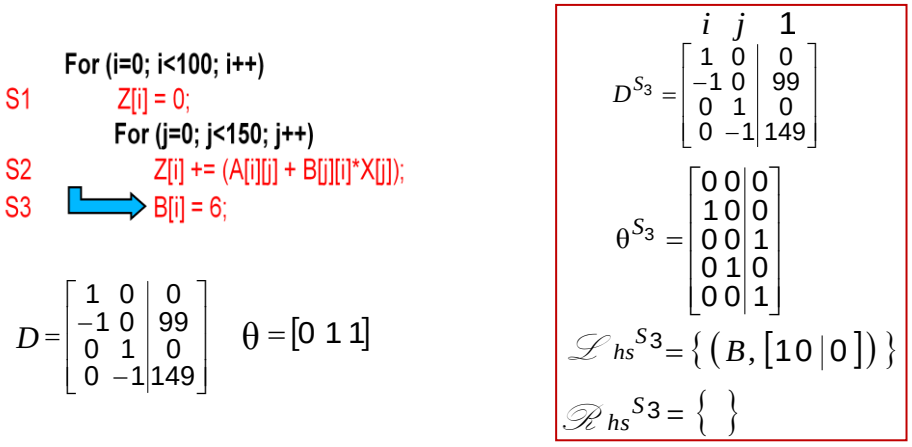


Figure 3.4 Polyhedral data structure for statement 1

And following figures are the construction process step by step.





As a result, when we trace all nodes in this nested loop by recursively applying these three steps in the algorithm of Figure 3.2, all statements' information we need is recorded.

3.3 Data Locality Transformation in Polyhedral Model

3.3.1 Loop Interchange (a.k.a. coalescing)

In this section, our goal is to change the order of the loops if the traversal way is different to the storage way. When we get a nested loop, we should analyze the array access in this loop to decide does it need to adopt the loop interchange.

Because C language belongs to row-major storage, we should make sure that the array access pattern in the nested loops would follow this rule. If we encounter a column-major array access, we should apply the loop interchange. There is an example to explain how to adopt the loop interchange shown in Figure 3.5.

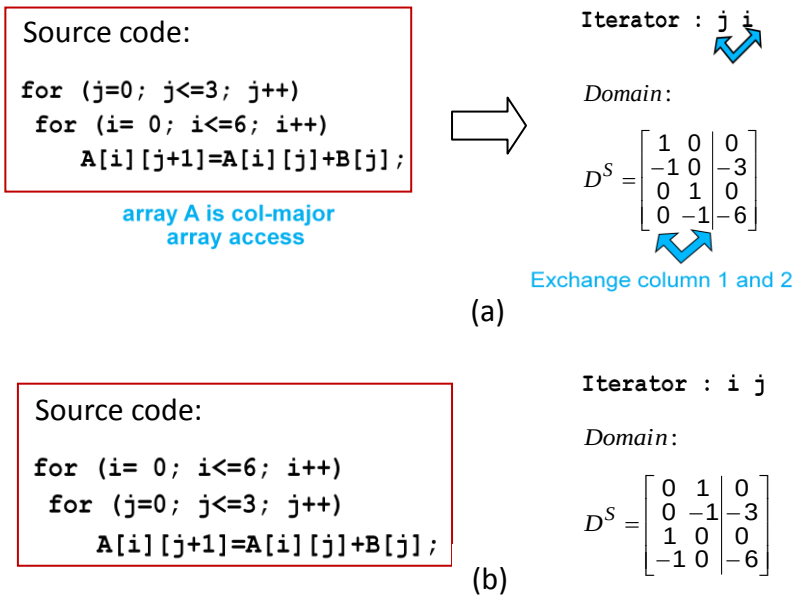


Figure 3.5 (a) Source code before loop interchange (b) Source code after loop interchange

The left-hand side of the Figure 3.5 (a) is the nested loops which we would analyze, and the right-hand side of the Figure 3.5 (a) is the corresponding data structure and processes which we should perform. Because the memory access of array A belongs to column-major traversal way which is violating the row-major storage way, we exchange the corresponding columns in the domain iteration and the iterator matrix. As a result, we could get the source code and the data structure after loop interchange in the Figure 3.5 (b).

3.3.2 Tiling (a.k.a. blocking)

Loop tiling is one of the loop optimizations for reducing the execution time of the nested loops which access a lot of array elements. Typically, the cache miss happened when too much other datum was accessed between the use and reuse of same specific data. It means that partitioning a loop's iteration domain into smaller tiles or blocks could ensure datum used in a nested loop sustains in the cache (or a faster level of the memory hierarchy). Thus fitting array elements of a tile into cache size could enhance the cache reuse frequency and eliminate cache size requirements.

To achieve that, we propose a mechanism to improve spatial data locality and temporal data locality in parallel programs. Spatial data locality refers to the use of data elements within relatively close storage locations. Accordingly, it could be improved by

evaluating the average memory cost per tiles. Temporal data locality refers to the reuse of specific data within relatively a short period of time. Therefore, it could be optimized by evaluating the datum reuse amount per tiles through analyzing array access within nested loops.

So far we have known that iteration-domain tiling (or blocking) is a well known loop transformation, but how to decide the tile sizes for a nested loop is a challenge. If the tile sizes are too large, the accesses of array elements within the nested loops would cause the cache miss frequently. If the tile sizes are too small, the improvement in data locality due to the loop tiling would not be significant and might have the extra computation overhead of inner tiled loops.

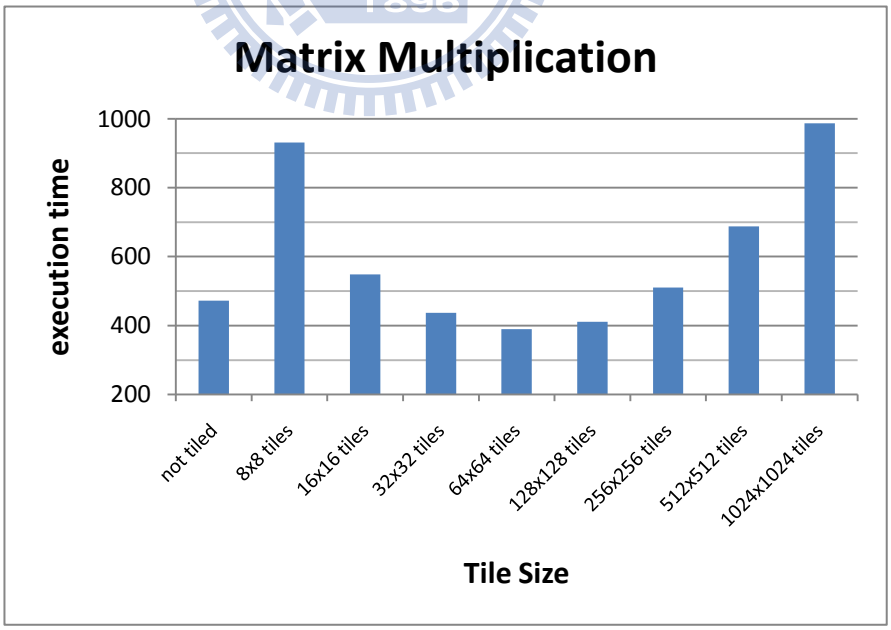


Figure 3.6 Performance of Matrix Multiplication on different tile size

On above Figure 3.6, we could realize that the different tile sizes affect performance

dramatically. Therefore, an appropriated tile size would lead a better cache behavior and a lower execution time.

In this thesis, we proposed a mechanism to choose an appropriated tile size from spatial and temporal analysis results. In the next two sections, we would introduce the two methods in detail. And the *tile sizes* for the nested loops, loop i and loop j which would be tiled respectively, are *Si* and *Sj*.

3.3.2.1. Spatial data locality analysis

Optimal tiling for spatial data locality refers to the problem of selecting the tile sizes that minimize the average memory cost which directly measure the execution time. In addition, we introduced the cost model which was proposed by Sarkar and Meggido, also used in the IBM XL FORTRAN compiler for evaluating tile sizes. This cost model is applicable to two or three level loops.

Our strategy is to estimate the average memory cost and record it for each tile size. The goal is to pick the tile sizes which minimize the memory cost, in other words the minimal total execution time. And the objective function used in the context of the tile size selection is all functions of the tile variables, cache capacity and cache line size, etc.

The memory cost of a tile(S) is calculated as $\mu_c \times DL(s) + \mu_p \times DP(s)$

, where $DL(s)$ is the estimated number of *distinct cache lines* accessed by a single tile and $DP(s)$ is the estimated number of *distinct pages* accessed by a single tile.

Given the hardware parameters: (1) the cache line size, L, (2) the effective cache size, ECS, (3) the TLB page size, TLB, and (4) the miss penalties of cache and TLB, μ_c and μ_p. Then we could calculate the memory cost of a tile by the previous hardware information to. The detailed formulation would be showed below.

calculate $\dfrac{\mu_c \times DL(s_i, s_j) + \mu_p \times DP(s_i, s_j)}{s_i s_j}$ per tile size

subject to $DL(s_i, s_j) \leq$ *Effective cache size (# lines of cache)*

$1 \leq s_i, s_j \leq$ *loop upper bound*

$s_i, s_j \in Z$

DL, DP : number of distinct cache lines and pages touched by a tile

μc, μp : cache and TLB miss penalties

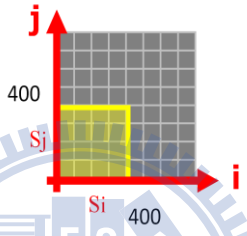
For calculating average memory cost, we could divide the memory cost to tile volume, which means *Si*Sj*. Then the SCL (Spatial_Coulddidate_List) would keep these average memory costs for each tile size. Accordingly, section 3.3.2.3 would select the tile size by referring SCL and TCL (Temporal_Coulddidate_List).

So far we have introduced the method to calculate the average memory cost for each tile size. Moreover, let's take an example to explain how it works. Figure 3.4 is a nested loop with two loops (loop i and loop j)

```
Source code :
real*8 A[400,400]
real*4 B[400,400],C[400,400]
for i = 1, 400
        for j = 1, 400
                A[k,i] = A[k,i] + B[i,j] x C[j,k]
```

Figure 3.7 Source code of spatial data locality analysis

The iteration domain may look like the gray grids in following figure. The range of dimension i is from 1 to 400, same as dimension j. And the tile size *Si*Sj* may be 8*8, 16*16, or 4*4 (such as yellow block) in the following figure.



Given the cache capacity is 4K bytes, hence, the cache capacity constraint could be written as: $DL(s_i, s_j) \le 4096$. The cache capacity constraint ensures that the same datum could be accessed within a tile. In addition, other hardware information would be μ_c =50 cycles, μ_p =260 cycles, L=128 bytes, L1 cache size=32K bytes, and it's column-major storage. These given information would vary on different machine.

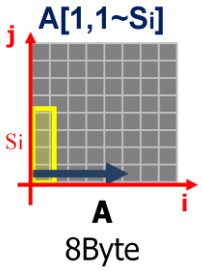
Hence, the number of distinct cache lines accessed by tile sizes of *Si*Sj* is estimated as follows:

$$DL(s_i, s_j) = s_i + \left\lceil \frac{4s_i}{128} \right\rceil s_j + \left\lceil \frac{4s_j}{128} \right\rceil \quad , \; 1 \le s_i, s_j \le 400$$

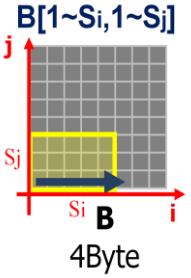
We could view the statement in this nested loop as

A[1,i] = A[1,i] + B[i,j] x C[j,1]

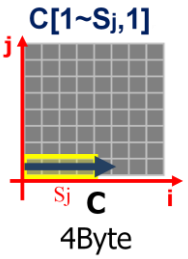
The first array access in the statement is A[1,i], and it would use elements from A[1][1~*Si*]. Graphic representation would show in the right-side figure. In this case, we need *Si* distinct cache lines to deal with all array accesses of array A.



The second array access is B[i,j], and it would use elements from A[1~*Si*][1~*Sj*]. Graphic representation would show in the right-side figure. In this case, we need $\left\lceil \frac{4s_i}{128} \right\rceil s_j$ distinct cache lines to deal with all accesses of array B.



The last array access is C[*Sj*,1], and it would use elements from A[1~ *Sj*][1]. Graphic representation would show in the right-side figure. In this case, we need $\left\lceil \frac{4s_j}{128} \right\rceil$ distinct cache lines to deal with all accesses of array C.



As a result, $s_i + \left\lceil \frac{4s_i}{128} \right\rceil s_j + \left\lceil \frac{4s_j}{128} \right\rceil$ distinct cache lines would be required within a single tile. That's how $DL(s_i, s_j)$ be calculated.

On the other hand, the number of distinct pages accessed by a tile of *Si***Sj* iterations are estimated as follows:

$$DP(s_i, s_j) = \left\lceil \frac{8 \times 400 s_i}{4096} \right\rceil + \left\lceil \frac{4s_i + 4 \times 400 s_j}{4096} \right\rceil + \left\lceil \frac{4s_j}{4096} \right\rceil \quad , \quad 1 \le s_i, s_j \le 400$$

The way to calculate in DP is similar to the way to calculate in DL, so we

would skip this detail process. As a result, the value of $\frac{\mu_c \times DL(s_i, s_j) + \mu_p \times DP(s_i, s_j)}{s_i s_j}$ would be stored in SCL (Spatial Candidate List) for each tile size.

3.3.2.2. Temporal data locality analysis

For each tile size in SCL, we analyze the temporal data locality as well. In other words, calculating the reuse times across the two-dimension iterations. If the indices of the array access belong to innermost two levels' variable (respectively i and j), we would calculate the reuse amount for each array by the following equation:

reuse amount = $\sum$ (Si-idiff)(Sj-jdiff)

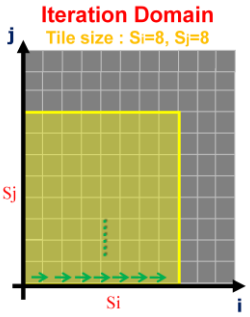
Sj means the first tile size, and Sj means the second tile size. The value of idiff represents how many across reuse happen when the j dimension is fixed. And the value of jdiff represents how many across reuse happen when the i dimension is fixed. There is a simple example to explain the idea shown as the following figure.

```
Source code :
for (m=0; m<100; m++)
    for (i=0; i<100; i++)
        for (j=0; j<100; j++)
            A[m][i]= A[m][i+1]+B[2][j];
            C[i][j]=B[2][j+2]+C[i+2][j+3];
```

If the tile size is 8*8, the reuse amount would be calculated like following equations.

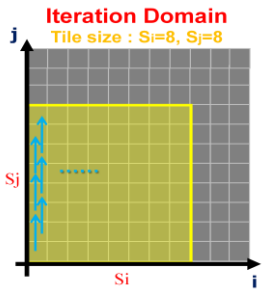
A[1][i], A[1][i+1] :

(Si-idiff)*Sj = (8-1)*8



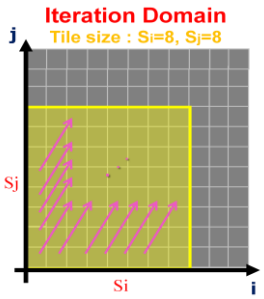
B[1][j], B[1][j+2] :

(Sj-jdiff)*Si = (8-2)*8



C[i][j], C[i+2][j+3] :

(Si-idiff)(Sj-jdiff) = 6*5



Therefore, the total reuse amount is 134.

As a result, we would store these values in TCL (Temporal Candidate List) for each tile size.

3.3.2.3. Combination with Spatial and Temporal data locality

In the section 3.3.2.1, we have calculated the average memory cost for each tile size and store in SCL (Spatial Candidate List). Moreover, we also have calculated the reuse amount for each tile size in the previous section 3.3.2.2 and store in TCL (Temporal Candidate List). Accordingly, we would select the final tile size by reference these two values.

The first problem we should conquer is that the smaller memory cost is better, but the larger reuse amount is better. Because there are the opposite directions, we need to normalize these two different kinds of values with different ways to make them in the same direction.

We normalize each average memory cost to the minimal average memory cost like the following equation:

$$NormSpat = \frac{TileSize_{\cos t} - \min TileSize_{\cos t}}{\min TileSize_{\cos t}} \qquad 0 \le NormSpat$$

The normalized spatial value would be a constant, which is larger than zero. Typically, the smaller value is the better.

On the other hand, we normalize each reuse amount to the maximal reuse amount like the following equation:

$$NormTemp = \frac{\max TileSize_{reuse} - TileSize_{reuse}}{\max TileSize_{reuse}} \qquad 0 \le TempSpat < 1$$

The normalized temporal value would be a constant, which is between one and zero. Typically, the value smaller is better, too.

The second problem we should conquer is how we prefer spatial analysis or temporal analysis. In order to choose how much percentage should take from spatial analysis, we imply the weight (α value) to our model. Therefore, we calculate the score to combine spatial and temporal analysis. The equation would be following:

$$score = \alpha \cdot NormSpat + (1-\alpha) \cdot NormTemp \qquad 0 \le \alpha \le 1$$

And thus, we choose the tile size which has minimal score to be the final tile size in tiling optimization.

3.4 Polyhedral Model to AST (Code Generation)

Code generation is to determine sets of statements whose execution order is interleaved and to create the appropriate loop nest structure. The basic mechanism is, starting from the list of polyhedral to scan, to recursively generate each level of the abstract syntax tree (AST) of the scanning code. As a result, we could output an abstract syntax tree that could be translated in high level language (for example, C language) or in a compiler's IR.

3.4.1 Extended Quiller´e et al. Algorithm

Earliest contributions [10-12] by Kelly et al. and Quiller´e et al. provide the way for generating efficient code for many statements with the overlapping polyhedral. The algorithm relies on polyhedral operations that could be implemented by library supporting, for example, PolyLib or PPL.

The notations which we would use are a polyhedron list ($\mathcal{T}_{S1}$, …. $\mathcal{T}_{Sn}$), the constraints set C, and the current dimension d. The constraints set labels all loop bounds and surrounding conditions. Accordingly, we would introduce the detail of the steps in

the algorithm.

Therefore, the extended Quiller´e et al. algorithm would introduce in the following Figure 3.8.

Input: a polyhedron list (Ts_1, …, Ts_n), constraints C, the current dimension d.

Output: the abstract syntax tree of the code scanning

1. Compute for each polyhedron T_{S_i}, its projection P_i onto the outmost d dimensions. Consider the new list of $P_i \longrightarrow T_{S_i}$
2. Start with $P_1 \longrightarrow T_{S_1}$ and $P_2 \longrightarrow T_{S_2}$, and compute $(P_1 - P_2) \longrightarrow T_{S_1}$, $(P_1 \cap P_2) \longrightarrow (T_{S_1}, T_{S_1})$ and $(P_2 - P_1) \longrightarrow T_{S_2}$, continue with $P_3 \longrightarrow T_{S_3}$.
3. Order according to lexicographical order.
4. for each $P \longrightarrow (T_{S_p}, \cdots, T_{S_q})$,
   4.1 Apply constraint $C \cap P$ to $T_{S_p}, \cdots, T_{S_q}$
   4.2 recurse with list $T_{S_p}, \cdots, T_{S_q}$ and dimension $d + 1$
5. For each $P \longrightarrow (\text{inside})$, apply steps 2 to 4 to *inside*.

Figure 3.8 Extended Quiller´e Algorithm

In other words, the most straightforward way to generate the resulting program is to apply the following simplified steps:

1. Create the scattering polyhedron for each statement by extending the iteration domain with the equalities.
2. Recursively project the previous polyhedron on the outermost dimensions to innermost dimensions to determine the span of each statement.
3. Recursively perform the intersection, difference and ordering of the previously projected polyhedral for all statements to distribute their iterations.

So far we have introduced the idea of the algorithm, but let us take an example for explain how this algorithm works. The original input code has three statements, named

S_1, S_2, and S_3. The Figure 3.9 is a polyhedral list, input of the algorithm. The list contains three polyhedrons.

$$T_{S_1} = \begin{cases} 1 \le i \le n \\ j = i \end{cases}$$
$$T_{S_2} = \begin{cases} 1 \le i \le n \\ i \le j \le n \end{cases}$$
$$T_{S_3} = \begin{cases} 1 \le i \le m \wedge m \ge n \\ j = n \end{cases}$$

Figure 3.9 A polyhedron list for code generation input

And the source code may be like any combination in Figure 3.9. The goal of this chapter is how to translate a polyhedral list to a correct source code which code size is small.

```
for i = 1 to n                    for i = 1 to n {
    if (i==j) S1                      if (i==j) S1
for i = 1 to n                        for j = i to n
    for j = i to n                        S2                  ……
        S2                            }
for i = 1 to m                    for i = 1 to m
    if (j==n) S3                      if (j==n) S3
```

Figure 3.10 Possible source codes of Figure 3.9

The initial domains to scan may look like below figure.

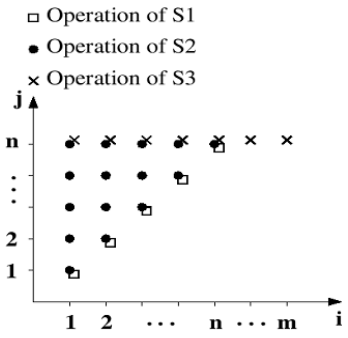


Figure 3.11 Initial domains of Figure 3.9

The first step in the algorithm is to compute projection ($\mathcal{P}$) for each polyhedral on

the outermost dimension. The projection list would like Figure 3.12.

Project onto first dimension - i

$P_1 = 1 \leq i \leq n$

$P_2 = 1 \leq i \leq n$

$P_3 = 1 \leq i \leq m \wedge m \geq n$

Figure 3.12 Projection list onto the first dimension.

The second step in the algorithm is to compute separate projections into a new list of disjoint polyhedral. Given a list of n polyhedron, start with the first two polyhedral $P_1 \rightarrow Ts_1$ and $P_2 \rightarrow Ts_2$ by computing $(P_1 - P_2) \rightarrow Ts_1$ (i.e. S_1 alone), $(P1 \cap P2) \rightarrow (Ts_1, Ts_2)$ (i.e. S_1 and S_2) and $(P_2 - P_1) \rightarrow Ts_2$ (i.e. S_2 alone). Then for the three resulting polyhedron, make the same separation with $P_3 \rightarrow Ts_3$ and so on. The following figures from Figure 3.13 to Figure 3.19 are detail processes.

$$\begin{cases} (P_1 - P_2) \rightarrow T_{S_1} & : \emptyset \\ (P_1 \cap P_2) \rightarrow (T_{S_1}, T_{S_2}) & : (1 \leq i \leq n) \\ (P_2 - P_1) \rightarrow T_{S_2} & : \emptyset \end{cases}$$

Figure 3.13 Operation on **statement 1 and 2**

$$\begin{cases} ((P_1 \cap P_2) - P_3) \rightarrow (T_{S_1}, T_{S_2}) & : \emptyset \\ ((P_1 \cap P_2) \cap P_3) \rightarrow (T_{S_1}, T_{S_2}, T_{S_3}) & : (1 \leq i \leq n) \\ (P_3 - (P_1 \cap P_2)) \rightarrow T_{S_3} & : (n+1 \leq i \leq m) \end{cases}$$

Figure 3.14 Operation on statement 1 and 2 **with statement 3**

To recurse with:

- $(1 \leq i \leq n) \rightarrow (T_{S_1}, T_{S_2}, T_{S_3})$
- $(n+1 \leq i \leq m) \rightarrow T_{S_3}$

Figure 3.15 Separate them into disjoint polyhedral, the result on first dimension

The result of projection and separation onto the first dimension domains may look like below figure.

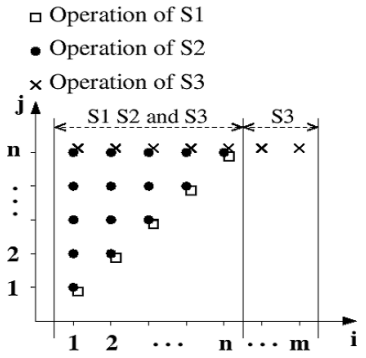


Figure 3.16 Separation result onto first dimension

So far we have projected the polyhedron onto the first dimension, i, then we separate them into two disjoint polyhedral.

Then we recurse on the next dimension, j, repeating the same previous process for each polyhedral list. The two polyhedral lists in first dimension are the following figures.

Recurse with:

$$T_{S_1} = \begin{cases} 1 \le i \le n \\ j = i \end{cases}$$

$$T_{S_2} = \begin{cases} 1 \le i \le n \\ i \le j \le n \end{cases}$$

$$T_{S_3} = \begin{cases} 1 \le i \le n \\ j = n \end{cases}$$

Recurse with: $T_{S_3} = \begin{cases} n+1 \le i \le m \wedge m \ge n \\ j = n \end{cases}$

There is only one polyhedron to consider (no change)

The first polyhedron list has many polyhedral could be separated, so we recurse this polyhedron list on next dimension, j. And the second polyhedron list has only one polyhedron which means that we don't need to recursively do it.

We apply the same previous process to the new polyhedron list. Computing separated projections into a new list of disjoint polyhedron. We start with the first two polyhedrons P_1 and P_2 by computing difference and intersection operations. Then join

the P_3 to compute the disjoint polyhedron. As a result, we could get the following figures.

$$\begin{cases} (P_1 - P_2) \rightarrow T_{S_1} & : \ \emptyset \\ (P_1 \cap P_2) \rightarrow (T_{S_1}, T_{S_1}) & : \ (1 \leq i \leq n \wedge j = i) \\ (P_2 - P_1) \rightarrow T_{S_2} & : \ (1 \leq i \leq n \wedge i + 1 \leq j \leq n) \end{cases}$$

Figure 3.17 Operation on **statement 1 and 2**

$$\begin{cases} ((P_1 \cap P_2) - P_3) \rightarrow (T_{S_1}, T_{S_2}) & : \ (1 \leq i \leq n \wedge j = i \wedge j \neq n) \\ ((P_1 \cap P_2) \cap P_3) \rightarrow (T_{S_1}, T_{S_2}, T_{S_3}) & : \ (1 \leq i \leq n \wedge i = n \wedge j = n) \\ (P_3 - (P_1 \cap P_2)) \rightarrow T_{S_3} & : \ \emptyset \end{cases}$$

$$\begin{cases} ((P_2 - P_1) - P_3) \rightarrow (T_{S_2}) & : \ (1 \leq i \leq n \wedge i + 1 \leq j \leq n - 1) \\ ((P_2 - P_1) \cap P_3) \rightarrow (T_{S_2}, T_{S_3}) & : \ (1 \leq i \leq n \wedge i + 1 \leq j \wedge j = n) \\ (P_3 - (P_2 - P_1)) \rightarrow T_{S_3} & : \ \emptyset \end{cases}$$

Figure 3.18 Operation on statement 1 and 2 **with statement 3**

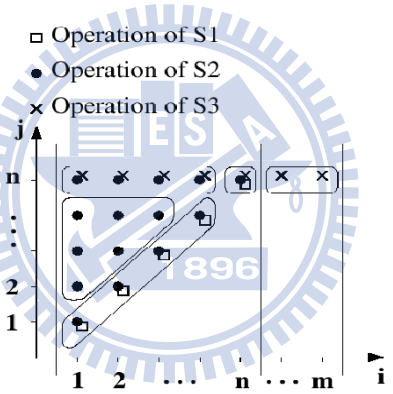


Figure 3.19 Separation result onto first and second dimensions

After all recursive steps to get the disjoint polyhedrons, we could return the final polyhedron list to generate each level of the abstract syntax tree or source code. For more detailed explanations, we refer the reader to the work of Quiller´e et al. and Bastoul [13]; here we only focus on a simplified example. In this section, we implement a mechanism to perform code generation process and translate them into AST.

Chapter 4 Experiment

In this chapter, the experiment environment and the simulation result are described. First, the experimentation environment is introduced. Then, the weight parameter, α value, would be determined. And finally, benchmark evaluation results discussed in Section 4.3. The evaluation results contain the reduction percentage of the execution time.

4.1 Environment

The CETUS compiler version 1.2 is used as our compiler infrastructure, because the CETUS compiler is easier to modify for programmers and provides many available passes such as OmpParser, LoopAnalysisPass, etc. For generating optimal OpenMP source code with data locality improvement, the CETUS compiler would be modified to analyze the nested loops with OpenMP annotations as input of our framework.

In our simulation, all of the programs are C source codes with OpenMP annotations. In addition, benchmarks used in this experiment were selected from NPB2.3-omp (NAS Parallel Benchmark) [14] and matrix-related parallel program, such as matrix-multiplication and matrix-transport. And the value of parameter α would be set from zero to one in which next section would discussed about.

4.2 Benchmark Evaluation Results

In this section, the α value in the *score* function is determined and our simulation results including the performance improvement. The simulation result of the benchmarks for our proposed methods are denoted as the T (named by temporal locality), the S (named by spatial locality), and the S+T (named by temporal and spatial locality). For comparison, the simulation results of the traditional multi-threads execution program (orig) are also depicted. And the environment parameter OMP_NUM_THREADS would set to eight for all programs which we simulated. Accordingly, eight threads would execute each program in parallel.

4.2.1 Parameter Determination

α value is the parameter to control the weight between SCL (Spatial Candidate List) and TCL (Temporal Candidate List) in our score function. If the α is equal to one, it means that the *score* of each tile size takes only the SCL into account; on the other hand, if the α is equal to zero, the *score* of each tile size takes only the TCL into account, apparently.

We have evaluated α value from 0 to 1 for our implementation, and the result is shown in Figure 4.3 to Figure 4.5. From the evaluation results, we observe that the α value would change scale slightly with different benchmark, because of the memory

access patterns and the cache behaviors differ from each benchmark. Based on our simulation results, the α value may be set to 0.25 for the proposed methods.

4.2.2 Performance improvement

The benchmarks which we would simulate are listing in Figure 4.1. The first five programs belong to NPB2.3-omp, and the other programs are matrix-related programs.

Benchmarks	Description
LU	LU Factorization
FT	3-D fast Fourier Transform (FFT)
MG	MultiGrid
SP	Scalar Pentadiagonal
BT	3-D compressible Navier-Stokes equations
transport	Matrix Transporation
omp_mm	Matrix Multiplication
mm_unroll	Matrix Multiplication with unroll j

Table 4.1 Description of benchmarks

LU uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system in 3-D by splitting it into block lower and upper triangular systems. **FT** contains the computational kernel of a 3-D fast Fourier Transform (FFT). **MG** uses a MultiGrid method to compute the solution of the 3-D scalar Poisson equation.

SP solves the finite differences that decouple the x, y and z dimensions. **BT** solves 3-D compressible Navier-Stokes equations. **Transport** and **omp_mm** are two widely used matrix operations. In addition, we modify the omp_mm to unroll the loop j four times for testing the temporal data locality (**mm_unroll**).

The experiment results for the different approaches (S, T, and S+T) discussed in section 3.3.2.1, 3.3.2.2 and 3.3.2.3 are presented as follows. We have simulated different α values for each benchmark which has temporal characteristic. The first experiment result is the comparison of original parallel code and tiling parallel code. The results of these alternatives are shown in the following Figure 4.1.

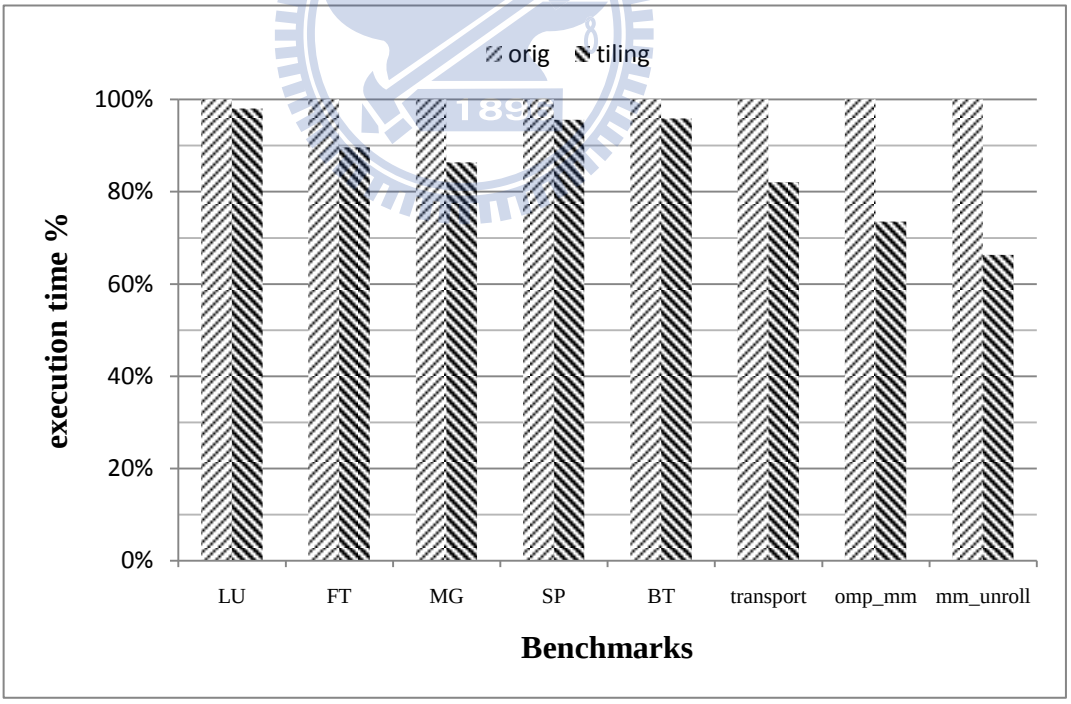


Figure 4.1 Evaluation results of origin and tiling

In Figure 4.1, the Y-axis indicates the execution time percentage, and the X-axis is the different benchmarks which be used. By using our proposed method, the execution

time percentage is improved mostly by about 33.7% compared to the traditional parallel codes. And the performance improvement is 14.1% in average.

Because the calculations are performed along the diagonal in LU, it does not utilize cache line well for either row-major or column-major storage. The performance almost does not improve.

The second experiment result is the comparison of different methods of data locality analysis in our work. The results of these alternatives are shown in the following Figure 4.2.

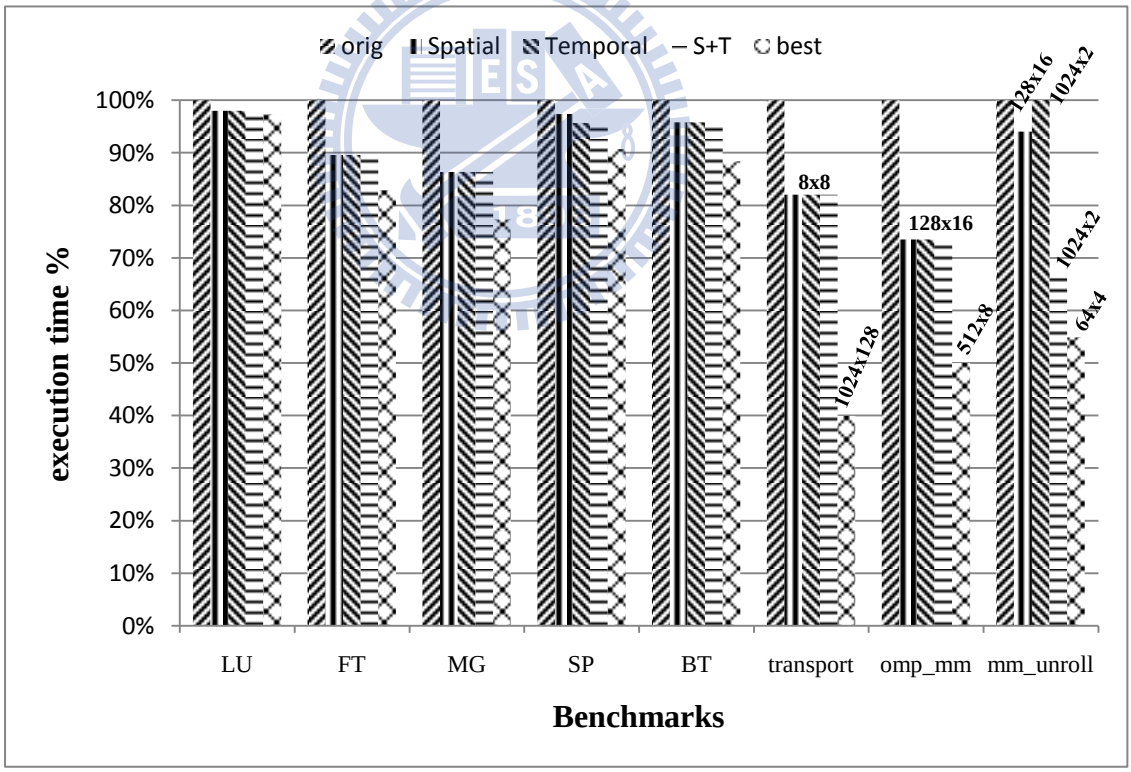


Figure 4.2 Evaluation results of origin, S, T, S+T and best

In Figure 4.2, the Y-axis indicates the execution time percentage, and the X-axis is the different benchmarks which be used. And the corresponding tile size of different data

locality analysis ways is marked above the execution time bar. The best approach means that the minimize execution time from all possible tile sizes would be choose for each nested loop. It is the ideal execution time which means the most performance improvement for all combinations of the tile sizes within each program.

We could observe the performance results by different data locality analysis ways of the mm_unroll parallel program which has temporal characteristic. The spatial method could improve 6% performance, and temporal method would loss performance significantly but hints the appropriated tile size preference. Therefore, we could select the proper α value to get the tile size for better performance result which reduces 33.7% execution time.

The experiment result in Figure 4.3 is one of the nested loops in SP program. The α value would not affect the performance, because S and T get the same tile size.

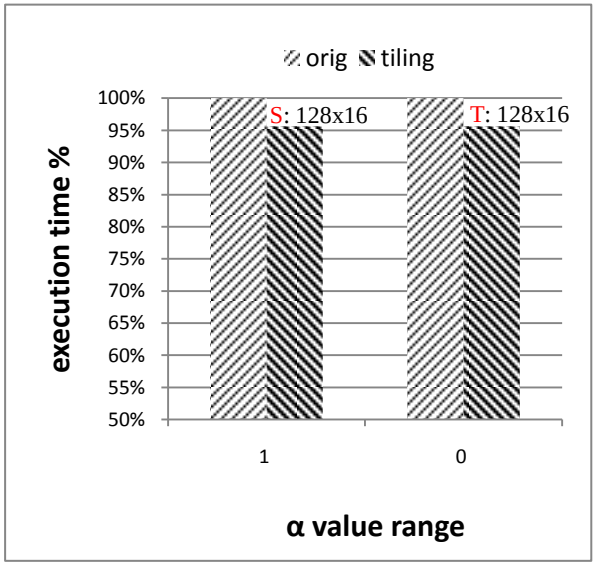


Figure 4.3 The evaluation result of different α values in SP

The experiment result in Figure 4.4 is another nested loops in SP program. Different α value would affect the performance, but it is difficult to get the performance improvement by loop tiling. Unfortunately, loop tiling not always works well.

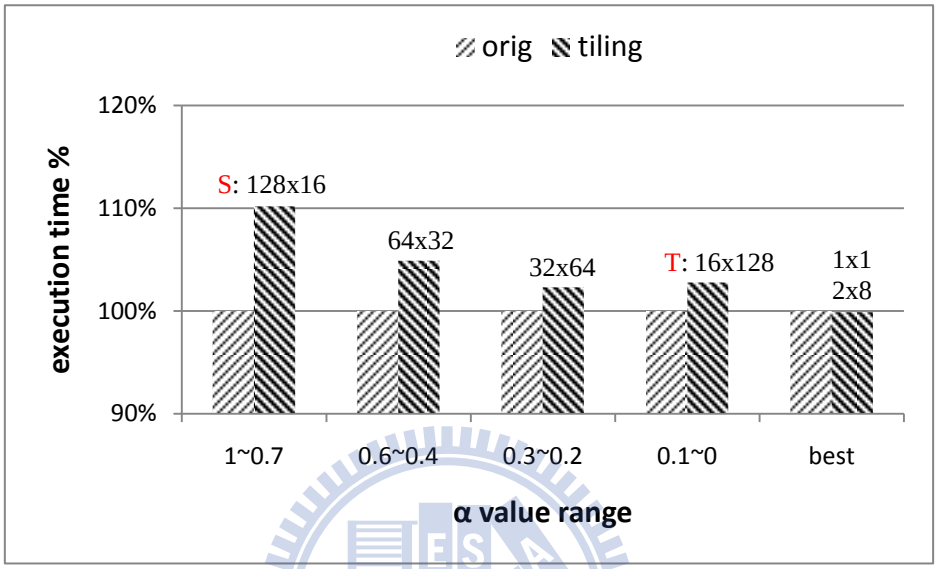


Figure 4.4 The evaluation result of different α values in SP

The last experiment result is the execution time for different α value in mm_unroll program. The results of these alternatives are shown in the following Figure 4.5.

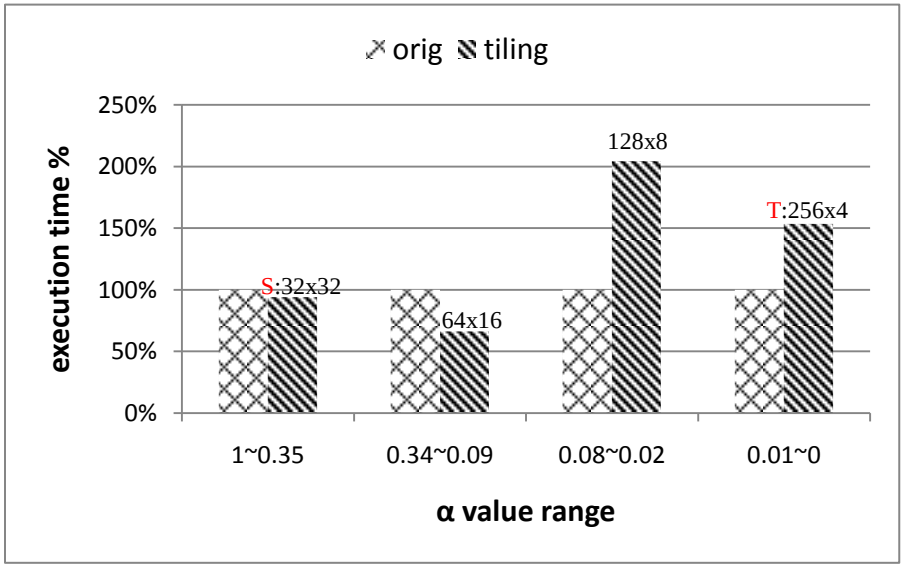


Figure 4.5 The evaluation result of different α values in mm_unroll

In Figure 4.5, the Y-axis indicates the execution time percentage, and the X-axis is the different α value which is chose. And the corresponding tile size of different α value is marked above the execution time bar. The performance would improve 33.7% execution time if an appropriated α value is set. And an improper α value may cause the performance degradation.

4.3 Summary for Simulation Results

In this chapter, we presented experiment results for each of our proposed methods. It is obvious that the approaches proposed in this thesis are usually effective for reducing execution time by using SCL and TCL for an appropriate α value.

By the statistics, the first data locality analysis method, SCL, could reduce 10.4% execution time in average. In addition, we have demonstrated the different results of performance improvement in different α values. In this thesis, the α value is set to 0.25 for performance consideration for programs. And the last S+T method could improve 33.7% performance at most, 14.1% in average.

Chapter 5 Conclusions and Future Works

In this chapter, the conclusions of this thesis are made first, and then the future works of this thesis are proposed.

5.1 Conclusions

In this thesis, we proposed a polyhedral framework to improve the data locality by modifying a current compiler backend. The framework included three parts of the processes which were translated a C source code with OpenMP annotations (AST) to polyhedral model, performed loop transformations, and translated it back to AST.

From the analysis result shown above, the performance increases 33.7% at most and 14.1% in average. But the performance doesn't improve significantly when the array sizes in those benchmarks are not very large. Because the cache behavior and the runtime variations of multiprocessors are usually unpredictable and difficult to analysis, the performance of our method could not achieve the best performance. For these reasons, the OpenMP threads number and other execution processes would affect the performance dramatically.

5.2 Future Works

The future works of this thesis could be put into three dimensions: to implement more transformations in polyhedral model, to consider the multiprocessors environment would cause what different effects, to modify the evaluation of tile sizes for getting a better performance improvement.

The first, we could implement other loop transformations[15-16] to make this polyhedral model framework more powerful. For example, loop fission, loop fusion and loop unrolling may be a choice to implement for improving the performance of the source code.

The second, executing program on multiprocessors has many unpredictable factors to affect the performance. Analyzing and observing the relationship between these factors may find out some optimization opportunities and modifications of cost model.

Finally, it may have chances to improve performance by modifying the evaluation of tile sizes, because of the previous simulation results and the relationship between multiprocessors factors. Hence, the performance improvement would work well in the future, regardless of the thread number which is executing the OpenMP source code and the resource competition.

References

[1] S. Girbal*, et al.*, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," Kluwer Academic Publishers 0885-7458, 2006.

[2] U. K. R. Bondhugula, "Effective automatic parallelization and locality optimization using the polyhedral model," Ohio State University, 2008.

[3] N. T. Vasilache., "Scalable Program Optimization Techniques In The Polyhedral Model," PhD thesis, 2007.

[4] H. Le Verge, "A Note on Chernikova's algorithm," ed, 1992.

[5] V. Loechner, "PolyLib - A library of polyhedral functions," http://icps.u-strasbg.fr/polylib/, 2000.

[6] R. B. a. P. M. H. a. E. Zaffanella, "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software systems," http://www.cs.unipr.it/ppl/, 2008.

[7] D. K. Wilde, "A Library for Doing Polyhedral Operations," December 2000 2000.

[8] e. a. P. Feautrier, "PipLib : Solving systems of affine (in)equalities," PRiSM, Versailles University2002.

[9] H. B. Chirag Dave, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, Samuel Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *IEEE Computer,* vol. 42, pp. 36-42, 2009.

[10] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," presented at the Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, 2004.

[11] C. Bastoul, "Efficient Code Generation for Automatic Parallelization and Optimization," in *ISPDC'2 IEEE International Symposium on Parallel and Distributed*

Computing, Ljubljana, Slovenia, Oct. 2003. , 2003, pp. 23-30.

[12] N. Vasilache*, et al.*, "Polyhedral Code Generation in the Real World," ed, 2006, pp. 185-201.

[13] C. Bastoul, "CLooG : Generating loops for scouldning polyhedra," PRiSM, Versailles University2002.

[14] M. F. a. J. Y. H. Jin "The OpenMP implementation of NAS parallel benchmarks and its performance.," 1999.

[15] J. R. Uday Bondhugula, P. Sadayappan, "PLuTo: A practical and fully automatic polyhedral program optimization system," *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08),* 2008.

[16] U. Bondhugula*, et al.*, "A practical automatic polyhedral parallelizer and locality optimizer," presented at the Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, Tucson, AZ, USA, 2008.