A Framework for Automatic OpenMP Code Generation

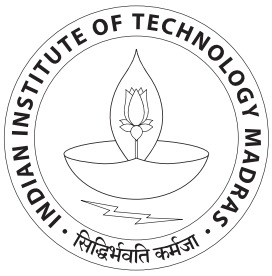
A Project Report

submitted by

RAGHESH A

in partial fulfilment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

April 2011

CERTIFICATE

This is to certify that the project report entitled **A Framework for Automatic OpenMP Code Generation**, submitted by **Raghesh A**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Shankar Balachandran
Research Guide
Assistant Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

The successful completion of this project would not have been possible without the guidance, support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to all those who were instrumental in the culmination of the project.

I am deeply indebted to my supervising guide Dr. Shankar Balachandran for his persistent encouragement and motivation, for his continual and creative feedback, for his stimulating suggestions in all time of work, and for his constructive criticism. His ingenious suggestions and thought provoking propositions have helped me widen my perspective on the subject matter of this report.

I offer my earnest gratitude to Tobias Grosser who has supported me throughout my project with his patience and knowledge. I gratefully acknowledge him for his advice, supervision, and crucial contribution, which made him a backbone of this project and so to this report. His involvement with his originality has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come. I am grateful in every possible way and hope to keep up our collaboration in the future.

A special thanks is also due to the faculty advisors, Dr N.S Narayanaswamy, Dr. C. Pandurangan and Dr. D. Janakiram who patiently listened, evaluated, criticized and guided us periodically. I extend my heartfelt thanks to Dr B. Ravindran,

Pramod C E and C K Raju for their valuable suggestions and care throughout the project.

Heartfelt love to my parents, brother and wife, the main pillars of my life, for being with me through thick and thin. Special thanks to Sunil, Jyothi Krishna, Balagopal, Ajeesh Ramanujan, Sunitha and Jignesh for their support and motivation.

ABSTRACT

KEYWORDS: Loop Transformation, OpenMP, Polyhedral Model, Vectorization, Autoparallelism

It is always a tedious task to manually analyze and detect parallelism in programs. When we deal with autoparallelism the task becomes more complex. Frameworks such as OpenMP is available through which we can manually annotate the code to realize parallelism and take the advantage of underlying multi-core architecture. But the programmer's life becomes simple when this is done automatically. In this report we present a framework for autoparallelism through Polly, a project to enable polyhedral optimizations in LLVM and the work done towards automatically generating OpenMP library calls for relevant parts of the code.

Various powerful polyhedral techniques exist to optimize computation intensive programs effectively. Applying these techniques on any non-trivial program is still surprisingly difficult and often not as effective as expected. Most polyhedral tools are limited to a specific programming language. Even for this language, relevant code needs to match specific syntax that rarely appears in existing code. It is therefore hard or even impossible to process existing programs automatically. In addition, most tools target C or OpenCL code, which prevents effective communication with compiler internal optimizers. As a result target architecture specific optimizations are either little effective or not approached at all.

Polly automatically detects and transforms relevant program parts in a language-independent and syntactically transparent way. Therefore, it supports programs written in most common programming languages and constructs like C++ iterators, goto based loops and pointer arithmetic. Internally it provides a state-of-the-art polyhedral library with full support for $\mathbb{Z}$-polyhedra, advanced data dependency analysis and support for external optimizers. Through LLVM, machine code for CPUs and GPU accelerators, C source code and even hardware descriptions can be targeted.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

LLVM	Low Level Virtual Machine
Polly	Polyhedral Optimization in LLVM
ClooG	Chunky Loop Generator
Isl	Integer Set Library
AST	Abstract Syntax Tree
SIMD	Single Instruction Multiple Data
CFG	Control Flow Graph
SCoP	Static Control Part
POCC	The Polyhedral Compiler Collection
GRAPHITE	GIMPLE Represented as Polyhedra Interchangeable Envelopes

CHAPTER 1

Introduction

1.1 Parallelism in programs

These days it is hard to find somebody using a single-core processor machine. With the help of multi-core and multi-processor machines it is possible to speed up the program by mapping the sections of the program to available processors. This is generally termed as parallelism in programs. It is very difficult to parallelize the entire program though. The degree of parallelism is limited by certain factors which is explained later in this section. In addition this section discusses various types of parallelism and make a comparison of various approaches towards parallelism which can be applied to programs.

1.1.1 Parallelism and locality

When there is a need for parallelism there is a need for interprocessor communication. So while optimizing programs for parallelism extreme attention should be given to minimize the communication overhead. We can minimize communication if the processor accesses recently used data. That is we need to improve data locality. Considering the performance of a single processor it is essential to extract more data locality which in turn increases the cache hits. While dealing with parallelism we need to be aware about the restrictions on the degree of parallelism

that can be extracted from a given program, which is well stated by **A**mdahl's law.

Amdahl's law states that, if f is the fraction of the code parallelized, and if the parallelized version runs on a p-processor machine with no communication or parallelization overhead, the speedup is given by,

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

For instance, if half the computation is sequential, the computation can only double in speed, regardless of the number of processors used. The speedup is a factor of 1.6 if we have 4 processors. So researchers keep on working for extracting more parallelism and thereby reducing the fraction of sequential computation.

1.1.2 Realizing parallelism

Some of the approaches to realize parallelism are explained in this section.

POSIX Threads/Pthreads

Pthreads provides a standard interface for performing multihreaded computation. Threads are subprocesses running with in a process. We can find many applications such as a web browser which can take advantage of multithreading. The efficiency of an application improves when it is designed with threads because they have their own stack and status. The overhead of creating a separate process can be avoided here. Resources like files are shared among threads. Though Pthreads are good alternatives for having multiple processes in a single processor machine it is very difficult to scale it to multi-core processors. Another limitation of Pthreads is programmers are required to deal with a lot of thread-specific code. The number

of threads required for a computation need to be hard corded which makes it less scalable.

OpenMP

In view of the shortcomings of POSIX threads there was an urge to formulate a new threading interface. The major objective was to overcome the burden of learning different ways for programming threads in different operating systems with in different programming languages. OpenMP is able to deal with this by a great extend. As the framework is evolved rather than its APIs, support for pragmas became the distinguished feature of OpenMP. The user has to specify only the blocks of code that need to be run as parallel. The compiler does the rest. It will take care of making the pragma annotated blocks into threads. Necessary APIs are inserted to map those threads into different cores. The example below shows usage of pragma.

```
#pragma omp parallel for
for (i = 1; i <= N; i++)
    A[i] = B[i] + C[i]
```

Another characteristic of OpenMP is that by disabling support for OpenMP the same program can be treated as single threaded. This enables easy debugging and makes the programmer's life easier.

If the developer needs more fine-grained control a small set of APIs are available in OpenMP. But in this case Pthreads could be the right choice because it provides a greater number of primitive functions. So if in applications in which threads require individual attention the appropriate choice would be Pthreads.

Ample care should be taken to ensure the correctness of the program while

using OpenMP pragmas. The following example illustrates that.

```
for (i = 0; i < 10; i++) {
  #pragma omp parallel for private(k)
  for(j = 0; j < 10; j++) {
    k++;
    A[i] += k;
  }
}
```

We get incorrect result if the data sharing attribute for the variable *k* is *private*. It should be *shared* to get the intended result.

1.2 Auto parallelization

The techniques described in the previous section relies heavily on manually identifying parallelism, which is not always a good approach. We can take the advantage of hardware support for parallelism only if the compiler has support for generating the parallel code. There are interfaces like OpenMP for developing parallel applications. But the user has to manually provide the annotations for it in the source code. This becomes a tedious task for the user and he has to ensure the correctness of the code too. This prompted researchers to explore mechanisms for finding out the parallel portions of the code without manual intervention.

It can be noticed that most of the execution time of a program is spend inside some for loop. Parallelizing compiler tries to split up a loop so that its iterations can be executed on separate processors concurrently. A dependency analysis pass is performed on the code to determine whether it can be safely parallelized. The

following example illustrates this.

```
for (i = 1; i <= N; i++)
    A[i] = B[i] + C[i]
```

The analysis detects that there is no dependency between two consecutive iterations and can be safely parallelized. Consider another example

```
for (i = 2; i <= N; i++)
    A[i] = A[i-1] * 2;
```

Here a particular iteration is dependent on previous one and so its not safe to parallelize. An intelligent compiler can convert this into parallel as follows.

```
for (i = 1; i <= N; i++)
    A[i] = A[1] * 2 ** (i - 1);
```

Detecting this kind of opportunities for parallelization and applying automatic transformation is a tedious task for existing compilers. A powerful mathematical model explained in the next section act as a helping hand for the compilers to do such transformations with some restrictions applied on the input.

1.3 The polyhedral model

In this model the program is transformed into an algebraic representation which can be used to detect data dependences. This representation is then converted in such a way that the degree of parallelism is improved. Polyhedral optimizations are used for many kind of memory access optimization by looking into the memory access pattern of any piece of code. Any kind of classical loop optimization

techniques like tiling can be used for this purpose. The model is explained in detail in Chapter 2.

1.4 LLVM

LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. One of the important features of LLVM is that the output of all the transformation passes have same intermediate representation(LLVM IR), which makes the programmer to analyze it with ease.

1.5 Polly and OpenMP code generation

The framework for automatic OpenMP code generation is implemented using, an open source[1] compiler optimization framework that uses a mathematical representation, the polyhedral model, to represent and transform loops and other control flow structures. It is an effort towards achieving autoparallelism in programs. The transformations are being implemented in LLVM(Low level virtual machine). Polly can detect parallel loops, issue vector instructions and generate OpenMP code corresponding to those loops. Polly try to expose more parallelism with the help of polyhedral model. A loop which does not look parallel can be transformed

[1] `http://llvm.org/releases/2.8/LICENSE.TXT`

to a parallel loop and these can be vectorized or parallelize using OpenMP. More details on LLVM and Polly can be found in Chapter 3.

1.6 Outline of report

The organization of this report is as described here. In Chapter 2 we describe the background required for understanding the polyhedral model. Chapter 3 deals with the internals of Polly - Polyhedral optimization in LLVM. Next chapter consists of the details of the workdone for OpenMP code generation in Polly. Then Chapter 5 explains the testing framework and shows the experimental results. And the last chapter has the list of future projects that can be done on Polly and we conclude with that.

CHAPTER 2

The Polyhedral Model

There are different types optimizations that can be performed on a program to improve its performance. The optimization can be made for finding data locality and hence extracting parallelism. Starting from the early history of programming languages the internal representation of program is done with Abstract Syntax Tree(AST). Though some elementary transformation can be performed on AST it is tough to carry out complex transformations like dependency analysis among statements inside a loop. Trees are very rigid data structures to do such transformations. In this chapter an extremely powerful mathematical model which puts together analysis power, expressiveness and flexibility is explained in detail.

2.1 Program transformations with polyhedral model

In this section some of the common program transformations which can be realized with the assistance of polyhedral model are explained. The polyhedral model is not a normal representation of programs when compared to the classical structure of programs(like AST) that every programmer is familiar with. But it is easier to do transformations smoothly in this model.

2.1.1 Transformation for improving data locality

The polyhedral model can detect common array accesses which improves the data locality. It is illustrated with a simple example.

```
for(i = 1; i <= 10; i++)
  A[i] = 10;


for(j = 6; j <= 15; j++)
  A[j] = 15;
```

The two loops will be represented by two polyhedrons and it can find the common array accesses starting from index 6 to 10 and the code can be transformed as follows.

```
for(i = 1; i <= 5; i++)
  A[i] = 10;
for(j = 6; j <= 15; j++)
  A[j] = 15;
```

2.1.2 Scalar expansion

```
for (i = 0; i < 8; i++)
  sum += A[i];
```

With the support of memory access transformation in polyhedral model this loop can be executed in parallel. It can be transformed to the code below where the scalar 'sum' is changed to array 'tmp'.

```
<create and initialize an array 'tmp' with size 4>
for (i = 0; i < 8; i++)
   tmp[i % 4] += A[i];
```

```
sum = tmp[0] + tmp[1] + tmp[2] + tmp[3];
```

With the help of some optimizer (like PLUTO[3]) the following code can be generated, where the outer loop is parallel.

```
parfor (ii = 0; ii < 4; ii++)
  tmp[ii] = 0;
  for (i = ii * 2; i < (ii+1) * 2; i++)
    tmp[ii] += A[i];
sum = tmp[0] + tmp[1] + tmp[2] + tmp[3];
```

2.2 Polyhedral representation of programs

The polyhedral model does its transformations based on linear algebra and linear programming. Certain parts of programs known as SCoPs(Static Control Part) are represented in this model. A program part that can be represented using polyhedral model is called SCoPs. Generally loops are the candidates for SCoPs. There are some restrictions to the set of statements in the section of code to be qualified as SCoP. Those are listed below.

- The set of statements in the loops should have bounds and conditionals having affine functions(linear combination with constant) of surrounding iterators and the parameters (constants whose values are unknown at compile time).
- There should be structured control flow.
- Side effect free(Only pure functions are allowed)

There are efforts to increase the application domain of polyhedral model [2] which shows most of the restrictions are artificial.

The representation of polyhedral model has three parts. Each of them is explained in detail with simple examples in the following sections.

2.2.1 Iteration domain

Consider the following loop.

```
for (int i = 2; i <= N; i++)
  for (int j = 2; j <= N; j++)
    A[i] = 10; // S1
```

Notice that the statement A[i] = 10 is denoted by S1. Even though this is a single statement, considering the loop as a whole it has several statement instances along the life time of the loop. Each statement instance has an **iteration vector** associated with it. In general the iteration vector for S1 is (i,j). To be more precise each statement instance has its own iteration vector. The set of all iteration vectors for a given statement is called **iteration domain** of that statement. Since the value of N is not known at compile time and the value is unchanged while runtime we call N as **parameter**. Hence the above loop is **parametric**. For the above loop we can write the iteration domain mathematically as

$$D_{S1} = \{(i, j)\, \epsilon\, Z^2 \mid 2 \leq i \leq N \wedge 2 \leq j \leq N\}$$

This is a subspace of Z^2. To get a better view this can be represented graphically as in Figure 2.1

Consider another example by just adding a conditional statement

```
for (int i = 2; i <= 6; i++)
  for (int j = 2; j <= 6; j++)
```

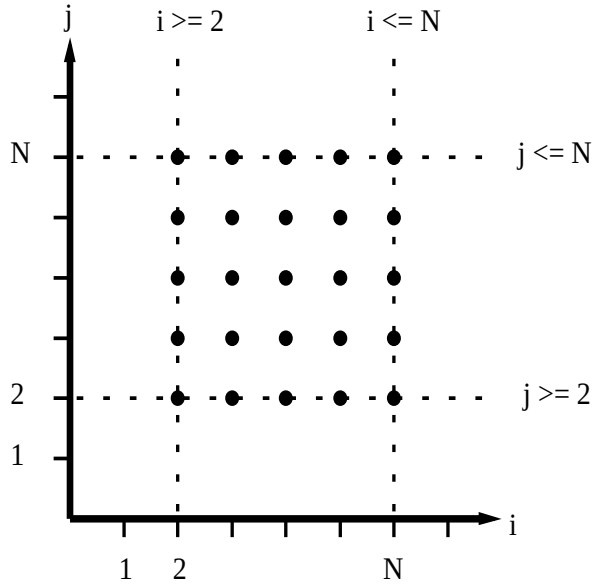


Figure 2.1: Graphical representation of iteration domain(S1)

```
if(i <= j)
  A[i] = 10; // S2
```

And the iteration domain here is as below and the corresponding graphical representation is shown in Figure 2.2

$$D_{S2} = \{(i,j)\, \epsilon\, Z^2 \mid 2 \leq i \leq 6 \wedge 2 \leq j \leq 6 \wedge i \leq j\}$$

Now we will consider a more complicated example where the iteration domain is subspace of Z^3.

```
for (int i = -3; i <= 3; i++)
  for (int j = -3; j <= 3; j++)
    for (int k = -3; k <= 3; k++)
      if (k <= j and (i + 4*k) <= 4*j)
        A[i][j] = 10; // S3
```

Iteration domain is as below and graphical representation is shown in Figure 2.3

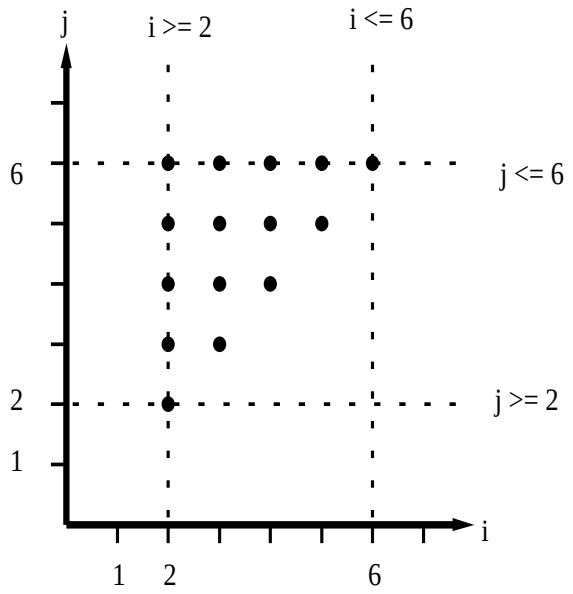


Figure 2.2: Graphical representation of iteration domain(S2)

$$D_{S3} = \{(i,j,k)\,\epsilon\, Z^3 \mid -3 \leq i \leq 3 \wedge -3 \leq j \leq 3 \wedge -3 \leq k \leq 3 \wedge k \leq j \wedge i + 4k \leq 4j\}$$

It can be seen that the iteration domain is specified by set of constraints. When those constraints are affine and depend only on the outer loop induction variables and parameters, the set of constraints defines a polyhedron (Z-polyhedron, polyhedron for short). Hence it has got the name Polyhedral Model.

2.2.2 Schedule

The iteration domain does not give any information on the order of statements to be executed. If there is no order specified among the execution of statements it means that all the statements can be executed in parallel. But due to data dependences this assumption may not be always true. So a method is devised to represent this order of execution which is called **scattering** function. There are many kinds of scattering in polyhedral model, as allocation, scheduling, chunking. For simplicity we are dealing only with **scheduling**. We are free to select any scheduling for better

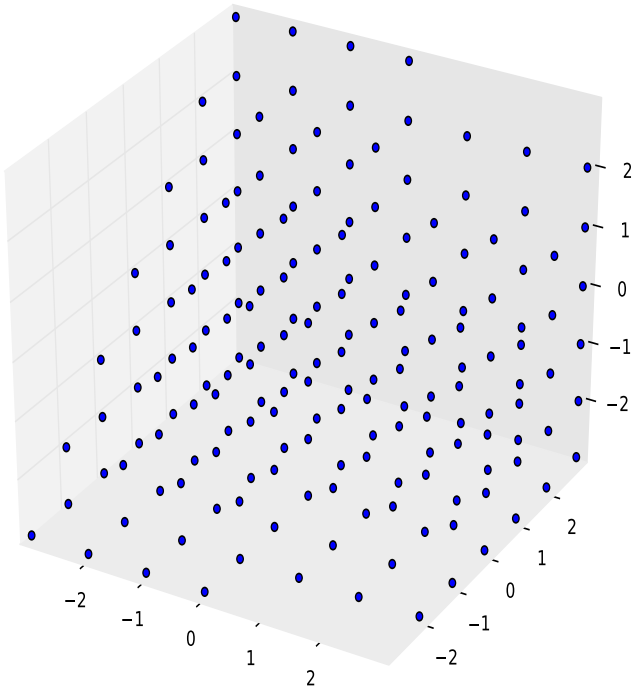


Figure 2.3: Graphical representation of iteration domain(S3)

transformation and hence better parallelism. Consider the following loop.

```
for (i = 0; i < 32; i++)
  for (j 0; j < 1000; j++)
    A[i][j] += 1 ; // S4
```

We can define a scheduling(scattering) function:

$$\phi_{S4}(i, j) = (i, j)$$

, which means that each iteration vector (i,j) in the iteration domain is associated with a logical date. That is the statement instances need to be executed in the lexicographical order of the logical date. Another possible scattering function would be:

$$\phi'_{S4}(i, j) = (j, i)$$

The code generated for this particular transformation is:

```
for (j = 0; j < 1000; j++)
  for (i = 0; i < 32; i++)
    A[i][j] += 1;
```

It can be observed that we just performed loop interchange. Consider another schedule as follows

$$\phi''_{S4}(i, j) = \{(i, jj, j) : jj \bmod 4 = 0 \wedge jj \leq j < jj + 4\}$$

Such a transformation can produce code which does strip mining, where we group a set of operations into different blocks. These block by block execution can improve data locality in some cases. The transformed code will look like:

```
for (j = 0; j < 1000; j++)
  for (ii = 0; ii < 32; ii += 4)
    for (i = ii; i < ii + 4; i++)
      A[i][j] += 1;
```

2.2.3 Access function

Consider a statement with array access A[i+j][i+N]. The **access function** corresponding to this can be written as:

$$F_A(i, j) = (i + j, i + N)$$

We can manipulate the access function for achieving better data locality and parallelism.

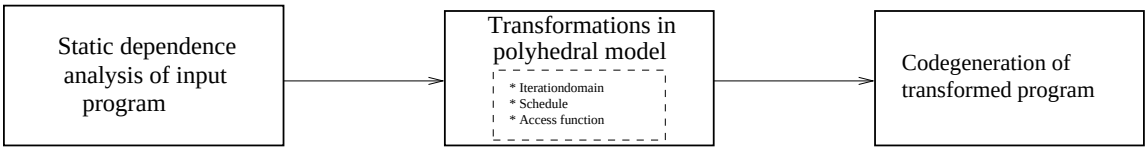


Figure 2.4: Transformation in polyhedral model

In general any optimization framework which makes use of polyhedral model will transform the code through the blocks shown in Figure 2.4. As the first stage dependence analysis is carried out followed by representing it by polyhedral model in terms of iteration domain, schedule and access function. This is further transformed for better optimization and code is generated for this.

CHAPTER 3

Polly - Polyhedral Optmizations in LLVM

This chapter deals with the internals of Polly - Polyhedral Optimization in LLVM[1].

3.1 Introduction to LLVM

LLVM(Low level virtual machine)[8] is a collection of reusable compiler toolchain technologies that allows sophisticated program transformations to be performed with much ease. The major feature of LLVM is its low level intermediate representation (LLVM-IR) which captures very minute details(like size and type of a variable) of the program code. Such details would be handy to implement effective optimizations. Tools are already available to convert programs written in popular programming languages like C, Python, Java, etc. to LLVM-IR and back. Polly operates on LLVM-IR, which increases the range of programming languages that gets benefited. A complete list of documents for LLVM is available here[2]

3.2 Introduction to Polly

Today, effective polyhedral techniques exist to optimize computation intensive programs. Advanced data-locality optimizations are available to accelerate sequential

[1] http://llvm.org
[2] http://llvm.org/docs/

programs [3]. Effective methods to expose SIMD and thread-level parallelism were developed.

Yet, the use of programming-language-specific techniques significantly limits their impact. Most polyhedral tools use a basic, language specific front end to extract relevant code regions. This often requires the source code to be in a canonical form, disallowing any pointer arithmetic or higher level language constructs like C++ iterators and prevents the optimization of programs written in languages like Java or Haskell. Nevertheless, even tools that limit themselves to a restricted subset of C may apply incorrect transformations, as the effects of implicit type casts, integer wrapping or aliasing are mostly ignored. To ensure correctness manual annotation of code that is regarded safe to optimize is often required. This prevents automatic transformations and consequently reduces the impact of existing tools.

With Polly we are developing a state-of-the-art polyhedral infrastructure for LLVM, that supports fully automatic transformation of existing programs. Polly detects and extracts relevant code regions without any human interaction. Since Polly accepts LLVM-IR as input, it is programming language independent and transparently supports constructs like C++ iterators, pointer arithmetic or goto based loops. It is built around an advanced polyhedral library and includes a state-of-the-art dependency analysis. Due to a simple file interface it is easily possible to apply transformations manually or to use an external optimizer. We use this interface to integrate Pluto [3], a modern data locality optimizer and parallelizer. With integrated SIMD and OpenMP code generation, Polly automatically takes advantage of existing and newly exposed parallelism.

This chapter focuses on concepts of Polly which are new or little discussed in

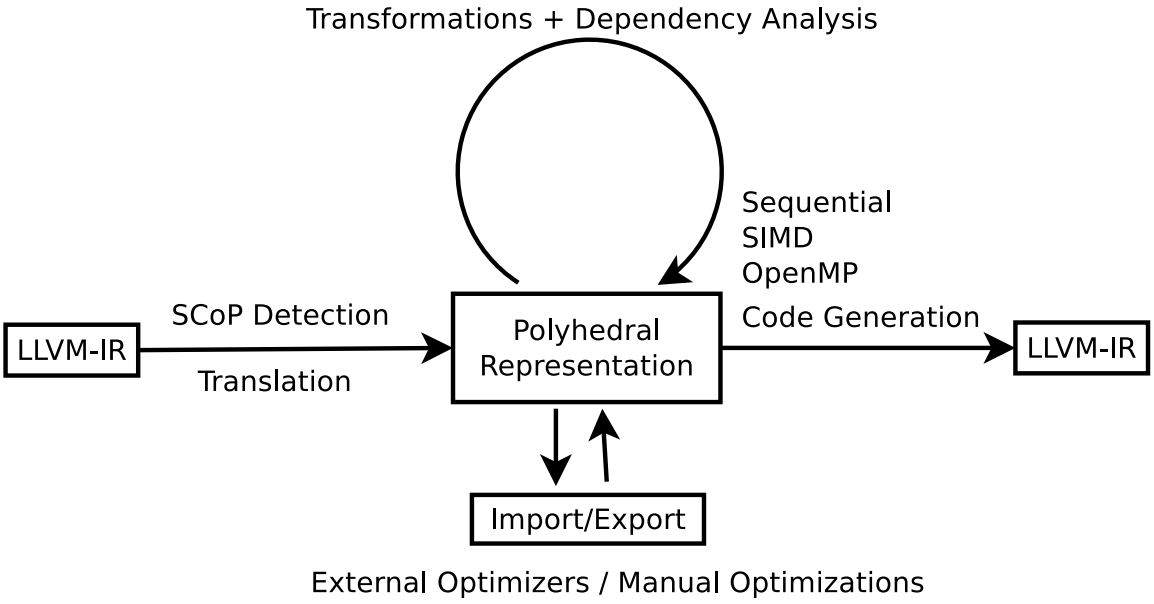


Figure 3.1: Architecture of Polly

the polyhedral community.

3.3 Implementation

Polly is designed as a set of compiler internal analysis and optimization passes. They can be divided into front end, middle end and back end passes. The front end translates from LLVM-IR into a polyhedral representation, the middle end transforms and optimizes this representation and the back end translates it back to LLVM-IR. In addition, there exist preparing passes to increase the amount of analyzable code as well as passes to export and reimport the polyhedral representation. Figure 3.1 illustrates the overall architecture.

To optimize a program manually three steps are performed. First of all the program is translated to LLVM-IR. Afterwards Polly is called to optimize LLVM-IR and target code is generated. The LLVM-IR representation of a program can be

obtained from language-specific LLVM based compilers. clang is a good choice. Polly also provides a `gcc` like user interface that is called `pollycc`.

3.3.1 LLVM-IR to polyhedral model

To apply polyhedral optimizations on a program, the first step that needs to be taken is to find relevant code sections and create a polyhedral description for them. The code sections that will be optimized by Polly are static control parts (SCoPs), the classical domain of polyhedral optimizations.

Region-based SCoP detection

Polly implements a structured, region-based approach to detect the SCoPs available in a function. It uses a refined version of the program structure tree described by Johnson [7].

A *region* is a subgraph of the control flow graph (CFG) that is connected to the remaining graph by only two edges, an entry edge and an exit edge. Viewed as a unit it does not change control flow. Hence, it can be modelled as a simple function call, which can easily be replaced with a call to an optimized version of the function. A *canonical region* is a region that cannot be constructed by merging two adjacent smaller regions. A region *contains* another region if the nodes of one region are a subset of the nodes of the other region. A tree is called *region tree*, if the nodes of it are canonical regions and the edges are defined by the contains relation.

To find the SCoPs in a function we look for the maximal regions that are valid

```
for (i = 0; i < n + m; i++)
  A[i] = i;
```

Figure 3.2: A valid syntactic SCoP. Not always a valid semantic SCoP

SCoPs. Starting from the outermost region, we look for canonical regions in the region tree that are valid SCoPs. In case the outermost region is a valid SCoP, we store it. Otherwise, we check each child. After analyzing the tree, we have a set of maximal canonical regions that form valid SCoPs. These regions are now combined to larger non-canonical regions such that the maximal non-canonical regions that form valid SCoPs are found.

Semantic SCoPs

In contrast to approaches based on the abstract syntax tree (AST), Polly does not require a SCoP to match any specific syntactic structure. Instead, it analyzes the semantics of a SCoP. We call SCoPs that are detected based on semantic criteria *semantic SCoPs*.

A common approach to detect a SCoP is to analyze an AST representation of the program, that is close to the programming language it is implemented in. In this AST control flow structures like for loops and conditions are detected. Then it is checked if they form a SCoP. Refer to Chapter 2 for the common restrictions that need to be met for a SCoP. There are various ways to extend this definition of a SCoP, which we did not include in this basic definition.

The detection of SCoPs as shown in Figure 3.2 with an AST based approach is easily possible, however as soon as programs become more complex and less

canonical difficulties arise. The AST of a modern language is often very expressive, such that there exist numerous ways a program can be represented. Sometimes different representations can be canonicalized. However, as soon as goto based loops should be detected, various induction variables exist or expressions are spread all over the program, sophisticated analyses are required to check if a program section is a SCoP. Further difficulties arise through the large amount of implicit knowledge that is needed to understand a programming language. A simple, often overlooked problem is integer wrapping. Assuming `n` and `m` are unsigned integers of 32 bit width, it is possible that $n + m < n$ holds. The upper bound in the source code must therefore be represented as $n + m \bmod 2^{32}$, but no polyhedral tool we know of models the loop bound in this way. Further problems can be caused by preprocessor macros, aliasing or C++ (operator) overloading. We believe even standard C99 is too complex to effectively detect SCoPs in it. Tools like PoCC, avoids this problem by requiring valid SCoPs to be explicitly annotated in the source code. However, this prevents any automatic optimization and significantly limits the impact of polyhedral techniques.

Fortunately, after lowering programs to LLVM-IR the complexity is highly reduced and constructs like implicit type casts become explicit. Furthermore, it is possible to run a set of LLVM optimization passes, that further canonicalize the code. As a result, an analysis that detects SCoPs based on their semantics is possible. LLVM-IR is a very low-level representation of a program, which does not have loops, but jumps and gotos and has no arrays or affine expressions, but pointer arithmetic and three address form operations. From this representation all necessary information is recomputed using advanced compiler internal analyses available in LLVM. Simple analyses used are loop detection or dominance infor-

```
int A[1024];
void pointer_loop () {
    int *B = A;
    while (B < &A[1024]) {
        *B = 1;
        ++B;
    }
}
```

Figure 3.3: Valid semantic SCoPs

mation to verify a SCoP contains only structured control flow. More sophisticated ones check for aliasing or provide information about side effects of function calls.

As Polly successfully recovers all necessary information from a low-level representation, there are no restrictions on the syntactic structure of the program source code. A code section is accepted as soon as the LLVM analyses can prove that it has the semantics of a SCoP. As a result, arbitrary control flow structures are valid if they can be written as a well-structured set of for-loops and if-conditions with affine expressions in lower and upper bounds and in the operands of the comparisons. Furthermore, any set of memory accesses is allowed as long as they behave like array accesses with affine subscripts. A loop written with `do..while` instead of `for` or fancy pointer arithmetic can easily be part of a valid SCoP. To illustrate this an example is shown in Figure 3.3.

3.3.2 Polyhedral model

The integer set library

Polly uses isl, an integer set library developed by Verdoolaege [10]. Isl natively supports existentially quantified variables in all its data structures; therefore, Polly

also supports them throughout the whole transformation. This enables Polly to use accurate operations on $\mathbb{Z}$-polyhedra instead of using polyhedra in the rationals as approximations of integer sets. Native support of $\mathbb{Z}$-polyhedra simplified many internal calculations and we expect it to be especially useful to represent the modulo semantics of integer wrapping and type casts.

Composable polyhedral transformations

Polly uses the classical polyhedral description [5] that describes a SCoP as a set of statements each defined by a domain, a schedule and a set of memory accesses. For more details refer to Chapter 2.

In contrast to most existing tools the domain of a statement cannot be changed in Polly. All transformations need to be applied on the schedule. There are two reasons for this. First, we believe it is conceptually the cleanest approach to use the domain to define the set of different statement instances that will be executed and to use the schedule for defining their execution times. As the set of different statement instances never changes there is no need to change the domain. The second reason is to obtain compositionality of transformations. As transformations on SCoPs are described by schedules only, the composition of transformations is simply the composition of the relations representing the schedules.

Export/Import

Polly supports the export and reimport of the polyhedral description. By importing an updated description with changed schedules a program can be transformed eas-

ily. To prevent invalid optimizations Polly automatically verifies newly imported schedules. Currently Polly supports the Scoplib exchange format, which is used by PoCC and Pluto [3]. Unfortunately, the Scoplib exchange format is not expressive enough to store information on existentially quantified variables, schedules that include inequalities or memory accesses that touch more than one element. Therefore, we have introduced a simple JSON[4] and isl based exchange format to experiment with those possibilities.

3.3.3 Polyhedral model to LLVM-IR

Polly uses CLooG [1] to translate the polyhedral representation back into a generic AST. This AST is then translated into LLVM-IR based loops, conditions and expressions.

Detecting parallel loops

Polly can detect parallel loops automatically and generates, if requested, thread-level parallel code by inserting calls to the GNU OpenMP runtime. This is targeted to automatically take advantage of parallelism present in the original code or exposed by previously run optimizers. To ensure correctness of generated code Polly does not rely on any information provided by external optimizers, but independently detects parallel loops. We present a novel approach how to detect them. For details refer to Chapter 4

3.4 Related work

The work in Polly was inspired by ideas developed in the Graphite project [9], yet Polly uses novel approaches in many areas. For instance, Graphite did not include a structured SCoP detection, even though currently a SCoP detection similar to the one in Polly is developed. Furthermore, Graphite works on the GCC intermediate representation, which is in several areas higher level than LLVM-IR, such that several constructs like multi-dimensional arrays are easily available. Internally Graphite still uses a rational polyhedral library and only in some cases relies on an integer linear programming solver. Graphite uses the classical parallelization detection before code generation and is not yet closely integrated with the OpenMP code generation. In contrast to Polly, it has been tested for several years and is reaching production quality.

The only other compiler with an internal polyhedral optimizer we know of is IBM XL/C. Unfortunately, we could not find any information on how SCoP detection and code generation is done. There exists a variety of source to source transformation tools such as Pluto [3], PoCC or LooPo.

CHAPTER 4

OpenMP Code Generation in Polly

4.1 Introduction

Transformations in Polly create loops that are executed in parallel, as if the user would have added some OpenMP pragmas. To achieve this, code generation needs to emit code that calls OpenMP library functions to be executed in parallel. The GNU OpenMP Library(libgomp) is used for this purpose. The dependency analysis module of Polly automatically detects parallel loops(SCoPs) and are given to OpenMP code generation module. Here we generate the required libgomp library calls. The generated code is similar to the one generated if the user have added OpenMP pragmas[1]. The following sections explain the steps taken towards generating the OpenMP code. The generated code is in LLVM IR format.

4.2 The framework

To get an understanding about OpenMP codegeneration in Polly consider the Figure 4.1.

A code written in any programming language(supported by LLVM) can be coverted into LLVM IR. Polly performs its transformations on LLVM IR and can

[1] `http://gcc.gnu.org/onlinedocs/libgomp/Implementing-FOR-construct.html`

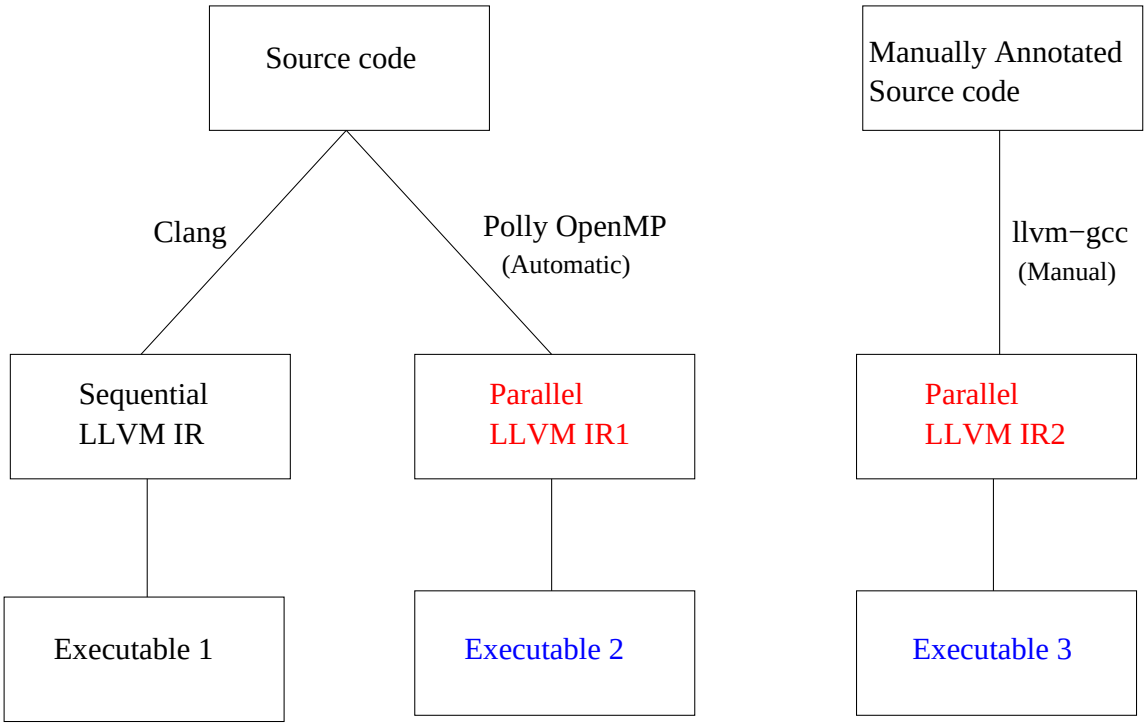


Figure 4.1: The framework

automatically generate IR (Parallel LLVM IR1) with OpenMP library calls. With manual annotation with OpenMP pragmas we can generate LLVM IR(Parallel LLVM IR2) using llvm-gcc. *Parallel LLVM IR1* and *Parallel LLVM IR2* will look similar and if we generated executable for both we are getting same performance. In short Polly with OpenMP support is as powerful as manual OpenMP annotated code with added advantage of automatic code generation.

4.3 Code generation pass in Polly

Refer to the detailed control flow in Polly in Figure 4.2. Each of the module in Polly is implemented as a LLMV pass[2]. We have to generate the OpenMP code in

[2] `http://llvm.org/docs/WritingAnLLVMPass.html`

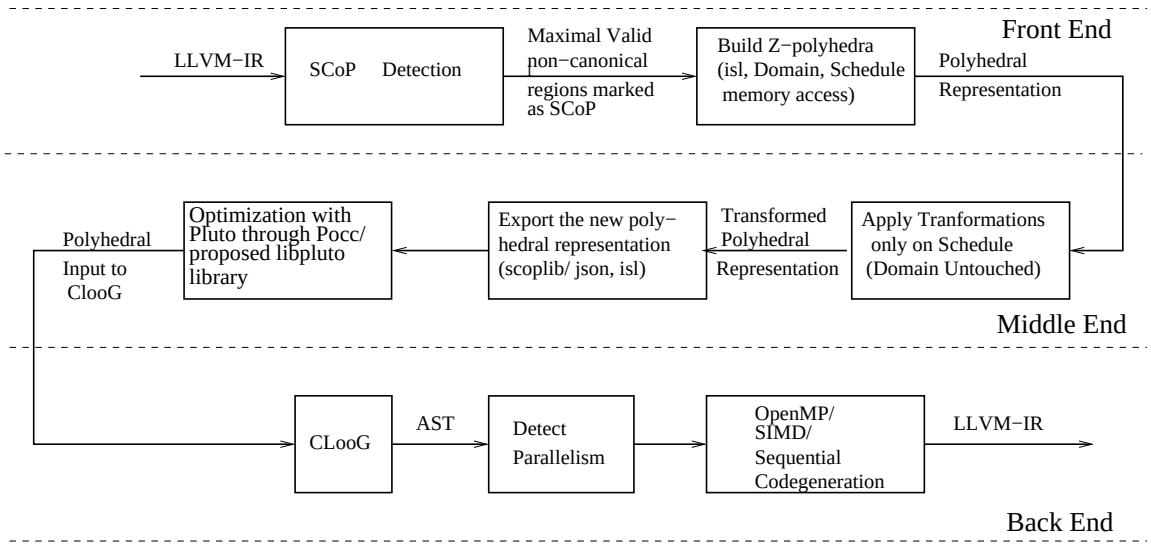


Figure 4.2: Detailed control flow in Polly

the code generation pass of Polly(CodeGeneration.cpp). LLVM does the magic of running this code generation pass for each of the detected SCoP. The runOnScop function initiates this. This function gather the required information by running all passes(ScalarEvolution, LoopInfo, CloogInfo, SCopDetection, etc.). The current pass can refer to the output of those passes as and when required.

While generating the code for a loop we check whether this loop is parallel or not. If it is parallel instead of generating the normal sequential code, loop is embedded in libgomp library calls. The approach for detecting parallelism is explained in the next section.

4.4 Detecting parallelism in Polly

A common approach to detect parallelism is to check before code generation, if a certain dimension of the iteration space is carrying dependences. In case it does not,

the dimension is parallel. This approach can only detect fully parallel dimensions. However, during the generation of the AST, CLooG may split loops such that a single dimension is enumerated by several loops. This may happen automatically, when CLooG optimizes the control flow. With the classical approach either all split loops are detected as parallel or no parallelism is detected at all.

The approach taken in Polly detects parallelism after generating the generic AST and calculates for each generated for-loop individually if it can be executed in parallel. This is achieved by limiting the normal parallelism check to the subset of the iteration space enumerated by the loop. To obtain this subset we implemented an interface to directly retrieve it from CLooG. As a result, we do not need to parse the AST to obtain it. With this enhanced parallelism check parallel loops in a partial parallel dimension can be executed in parallel, even though there remain some sequential loops. This increases the amount of parallel loops that can be detected in unoptimized code and removes the need for optimizers to place parallel and sequential loops in different dimensions.

Polly automatically checks all generated loops and introduces OpenMP parallelism for the outermost parallel loops. By default it assumes parallel execution is beneficial. Optimizers that can derive that for some loops sequential execution is faster may provide hints to prevent generation of OpenMP code. Polly could incorporate such hints during code generation easily, as they do not infect the correctness of the generated code.

4.5 Generating OpenMP library calls

Typically when a user want to run a particular section of the code in parallel he/she annotate the code with OpenMP pragmas. The compiler will then convert this pragmas into the corresponding library calls. In Polly the approach taken is to generate these calls automatically when a loop is detected as parallel. Consider the for loop below to have a basic understanding about what is to be done.

```
for (int i = 0; i <= N; i++)
  A[i] = 1 ;
```

This is detected as a parallel and given for OpenMP code generation. Here the following sequence of GOMP library calls with proper arguments and return types(signature) has to be generated in LLVM IR format. A general outline of the steps are given here and we enter into the implementation details.

- GOMP_parallel_loop_runtime_start
- subfunction
- GOMP_parallel_end

The control flow graph corresponding to the previous example is shown in Figure 4.3 The code for body of the for loop is generated inside the subfunction which has the following GOMP library calls to achieve the necessary parallelism.

- GOMP_loop_runtime_next
- GOMP_loop_end_nowait

The signature and descriptions of each of the above functions can be found in in libgomp manual[3].

[3] http://gcc.gnu.org/onlinedocs/libgomp/

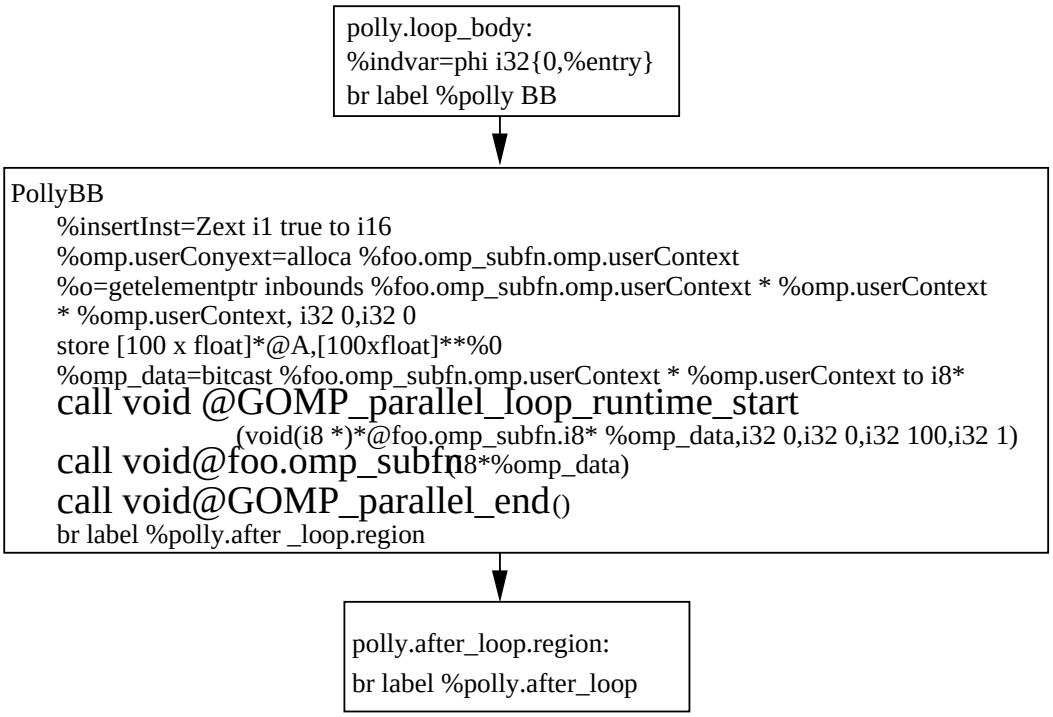


Figure 4.3: CFG showing sequence of OpenMP library calls

The very first step is to create the prototype for each of the functions in LLVM-IR format[4]. Code for generating prototype for "GOMP_parallel_end" is given below.

```
Module *M = Builder.GetInsertBlock()->getParent()->getParent();
LLVMContext &Context = Builder.getContext();
if (!M->getFunction("GOMP_parallel_end")) {
  FunctionType *FT = FunctionType::get(Type::getVoidTy(Context), false);
  Function::Create(FT, Function::ExternalLinkage, "GOMP_parallel_end", M);
}
```

The first line gets the global module(M) where we store the information about the function to be created. The 'Builder' function gives the current place to insert instructions. The 'FunctionType::get' method puts the types of arguments and return type and 'Function::Create' create and insert the prototype into the code. In

[4] http://llvm.org/docs/tutorial/LangImpl3.html#funcs

a similar manner all the prototypes for other functions are created.

The next step is to insert calls to the library calls. These calls replaces the sequential code for the original loop and the body of the loop is embedded in the OpenMP subfunction which is going to be exectuted as an OpenMP thread. Here is the code to create call to ”GOMP_parallel_end”:

```
Function *FN = M->getFunction("GOMP_parallel_end");
Builder.CreateCall(FN);
```

The next step which is more interesting is to create the body of the subfunction which involves the difficult process of linking the basic blocks in proper manner. Here is the code to generate an empty body for the subfunction:

```
LLVMContext &Context = FN->getContext();
// Create a new basic block to start insertion into.
BasicBlock *BB = BasicBlock::Create(Context, "entry", FN);
// Store the previous basic block.
BasicBlock *PrevBB = Builder->GetInsertBlock();
// Add the return instruction.
Builder->SetInsertPoint(BB);
Builder->CreateRetVoid();
// Restore the builder back to previous basic block.
Builder->SetInsertPoint(PrevBB);
```

This is just a basic code to create basic block ’BB’ with label ’entry’ The ’SetInsertPoint’ function can be used to insert code in a particular basic block. The complete control flow graph for the subfunction body of a typical for loop will look like Figure 4.4. In the basic block labelled ’omp.checkNext’ we call ”GOMP_loop_runtime_next” which finds the upper and lower bound of each OpenMP thread. In the basic block with label ’polly.loop_body’ we generate the sequential code for the loop.

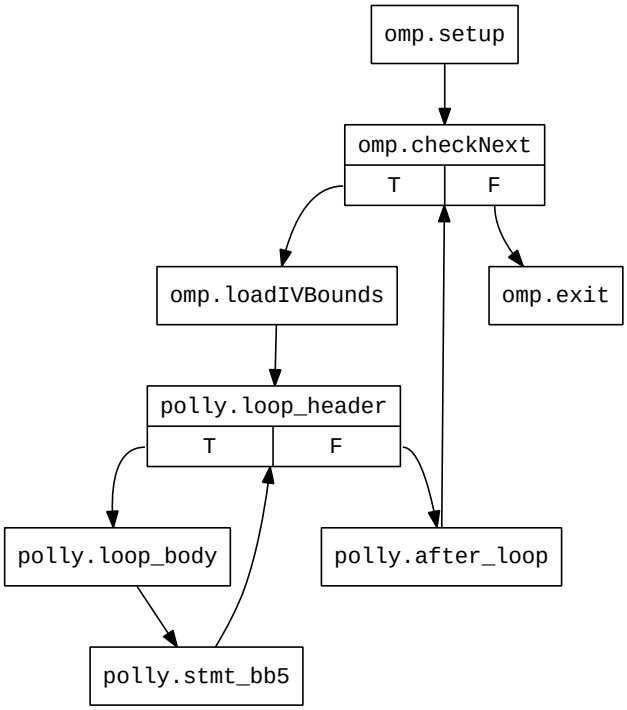


Figure 4.4: CFG showing basic blocks in the subfunction body

4.6 Support for inner loops

So far OpenMP code created apply only for outermost loops, which is detected as SCoP. Next step is to do it for inner loops. Due to dependency issues the outer loop is not detected as SCoP, but innerloop can be safely parallelized as in the following example.

```
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    A[i][j] = A[i-1][j] + B[i-1][j];
```

Those loops need the values of the surrounding induction variables and parameters in the OpenMP subfunction. We need to pass the values of the outer induction variables in a structure to the subfunction. All the required variables were already available in a data structure used by Polly. We just needed to copy

those into the body of the subfunction so that it can refer those whenever needed.

4.7 Dealing with memory references

```
#define N 10
void foo() {
  float A[N];
  for (int i=0; i < N; i++)
    A[i] = 10;
  return;
}
```

Consider the above code segment. The 'for' loop will be detected as parallel by Polly and will be embedded in the body of the OpenMP subfunction. But it accesses a non-global array 'A' and so accessing the same will not be possible inside the subfunction. The approach for solving this issue is explained below.

4.7.1 Adding memory references

The base addresses of all memory references made by a statement is available in each statement instance. Prior to creating the body of the subfunction we add all these base addresses are added into the same data structure where we stored the induction variables and parameters. And then it is added to the subfunction structure.

4.7.2 Extracting memory references

Inside the body of the subfunction the base addresses are extracted from the subfunction structure and a new LLVM load instruction is created for each. The new base addresses mapped to the old addresses so that any future references are made on the new addresses.

4.8 Enabling OpenMP code generation in Polly

Refer to Appendix A for setting up the environment for Polly. Here we describe the steps to generate OpenMP code in Polly. The easiest way to create the executable for a program with OpenMP support is by using the 'polycc' utility which is available in 'utils' directory under Polly source tree. Suppose you want to build a C file named 'a.c' issue the commands

```
export LIBPOLLY=<path to cmake>/lib/LLVMPolly.so
pollycc -fpolly -fparallel a.c
```

Polly is built as a shared library as LLVMPolly.so.

If we need more debugging options we can use the 'opt' command, which is the optimizer command for LLVM.

```
# Generate the LLVM-IR files from source code.
clang -S -emit-llvm a.c
alias opt="opt -load $LIBPOLLY
# Apply optimizations to prepare code for polly
opt -S -mem2reg -loop-simplify -indvars a.c -o a.preopt.ll
```

```
# Generate OpenMP code with Polly
opt -S -polly-codegen -enable-polly-openmp a.preopt.ll -o a.ll
# Link with libgomp
llc a.ll -o a.s
llvm-gcc a.s -lgomp
```

4.9 OpenMP testcases

Polly follows the LLVM testing infrastrcutre[5] to add regression testcases. Testcases for OpenMP are added into 'test/Codegen/OpenMP' directory under Polly source tree. The tests can be carried out by just issuing the command 'make polly-test' from the 'cmake' build directory.

[5] http://llvm.org/docs/TestingGuide.html

CHAPTER 5

Testing With PolyBench

This chapter deals with the experiments performed and the results obtained for our framework. The performance of OpenMP code generated by Polly compared with that of the code generated by various compilers on various machines. In the first section a simple program is tested to show that we get similar performance as that of a program having manual OpenMP annotations. Then in the next section we deal with test cases available in the PolyBench benchmark.

5.1 A simple test case

The following loop is tested on different machines and the results are shown in the Table 5.1.

```
for (i = 0; i < 1024; i++) {
  for (j = 0; j < 5000000; j++)
    A[i] += j;
}
```

The comparison is made in four different machines with the code compiled with

- Clang[1] which generates only serial code (Serial Execution).
- Polly with OpenMP enabled which generates necessary OpenMP code automatically (Automatic Parallelization).

[1] http://clang.llvm.org

- GCC with OpenMP enabled. Here the user have to give the OpenMP pragmas manually (Manual Parallelization).

	Serial Execution	**Automatic Parallelization(Polly)**	**Manual Parallelization(GCC)**
Intel Core 2 Duo(32 Bit OS)	9.509s	4.852s	4.835s
Intel Core 2 Duo(64 Bit OS)	6.40s	3.32s	3.50s
Intel Core i5(64 Bit OS)	6.96s	3.78s	3.75s
AMD Engineering Sample(24 Core)(64 Bit OS)	17.039s	0.757s	0.796s

Table 5.1: Performance Comparison

It can be observed that when OpenMP is enabled in Polly we are getting a performance almost similar to GCC with OpenMP pragmas provided by the user manually, which is the expected result. More speedup is obtained in the 24 Core machine.

5.2 PolyBench

The framework is tested with polyBench 1.0[2] and the results are shown in the next section. PolyBench is a set of computationally intensive programs often used in the polyhedral community. There are benchmarks from linear algebra, datamining, stencil computation and solver and manipulation algorithms operating on matrices. On those benchmarks Polly extracts the relevant SCoPs and optimizes them automatically.

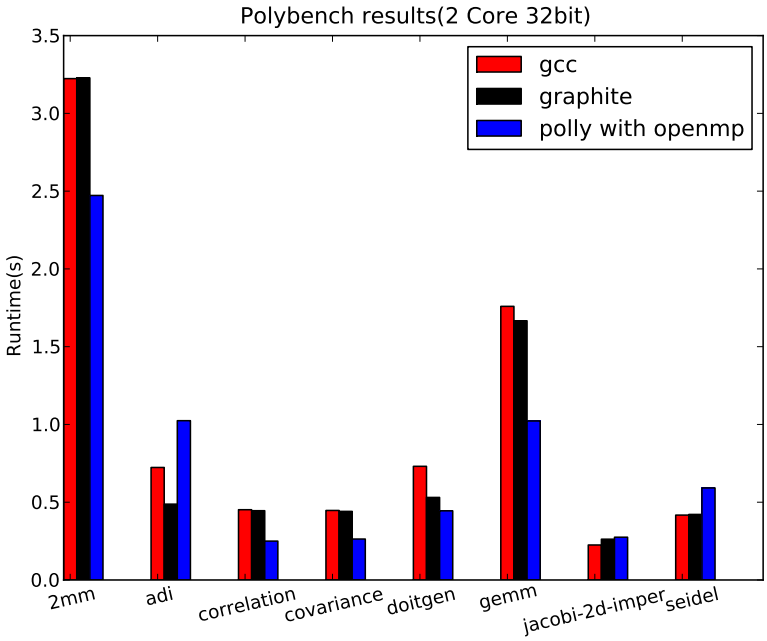


Figure 5.1: Performance comparison(2 core 32 bit)

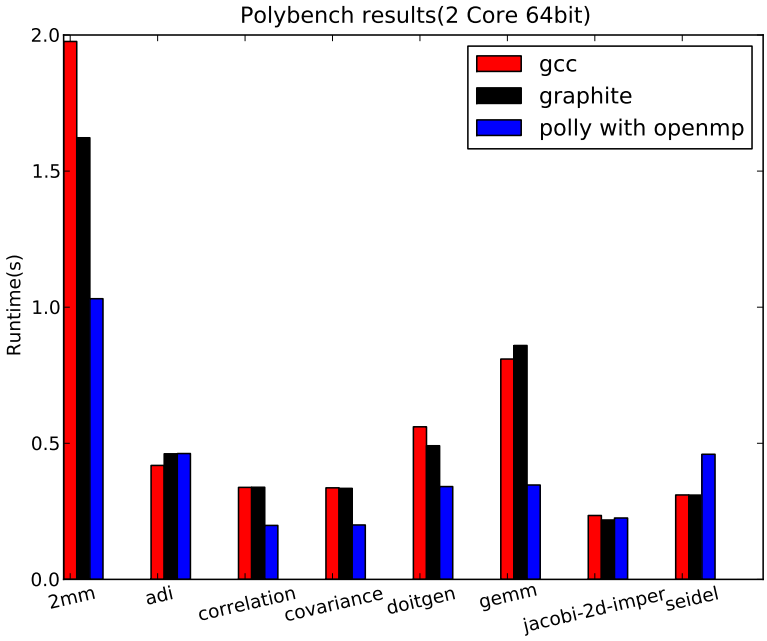


Figure 5.2: Performance comparison(2 core 64bit)

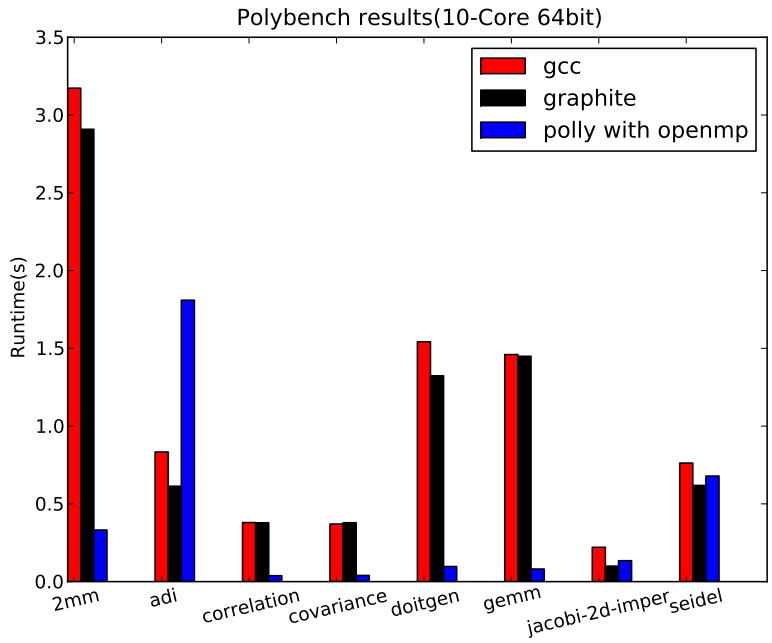


Figure 5.3: Performance comparison(10-core 64 bit)

5.3 Experimental results

The OpenMP code generated by Polly is compared with gcc and graphite[9]. With gcc we make a comparison with serial execution and with graphite we make comparison with an existing autoparallelization framework, which is also based on polyhedral model. The tests are carried out in 3 different machine with the following configurations

- Intel Core 2 Duo with 32 Bit OS
- Intel Core 2 Duo with 64 bit OS
- 10-Core AMD Engineering Sample with 64 Bit OS

The 10-core machine is part of GCC compile farm[3]. The GCC Compile farm project maintains a set of machines of various architectures and provides ssh

[2] http://www-roc.inria.fr/~pouchet/software/polybench/
[3] http://gcc.gnu.org/wiki/CompileFarm

access to free software developers, GCC and others. Once the account application is approved, we get full ssh access to all the farm machines. Then we are free to install any packages and test our work. The only prerequisite to get access is that we should be an active contributer for at least one free software project.

The script for testing is given below and the results are shown in the graphs in Figures 5.1, 5.2 and 5.3.

```
# serial
gcc -I utilities utilities/instrument.c -DPOLYBENCH_TIME    \
                        -DPOLYBENCH_DUMP_ARRAYS -O3 $1 -lm
# Autopar with graphite
n = 4 # n = 2 for 2 core, n = 10 for 10-core
gcc -I utilities utilities/instrument.c -DPOLYBENCH_TIME    \
            -DPOLYBENCH_DUMP_ARRAYS -O3 -floop-interchange  \
            -floop-block -floop-parallelize-all             \
            -ftree-parallelize-loops=$n $1 -lm
# Autopar with polly OpenMP
pollycc -fpolly -fparallel -I utilities utilities/instrument.c \
              -DPOLYBENCH_TIME -DPOLYBENCH_DUMP_ARRAYS  $1 -lm
```

While we look into the results it can be observed that Polly with OpenMP support shows nice performance other than the benchmarks 'adi' and 'seidel'. The reason for this is, due to dependences, the parallelism detection algorithm available in Polly is not able to detect the kernel of these testcases as parallel. Consider the kernel of 'seidel' given below.

```
for (t = 0; t <= tsteps - 1; t++)
  for (i = 1; i<= n - 2; i++)
    for (j = 1; j <= n - 2; j++)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
```

```
+ A[i][j-1] + A[i][j] + A[i][j+1]
+ A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])/9.0;
```

This has some loop carried dependences due to which Polly fails to detect parallelism. In this case Polly can get help from PLUTO optimizations. PLUTO can transform the loop in a way that some parallelism can be extracted. It is observed that OpenMP code is generated in this way. But the result was not improved at least in case of 'seidel'. It was improved when the OpenMP scheduling policy is changed. The default scheduling policy set while generating OpenMP code is 'runtime', using which the user can provide one of the three scheduling algorithms('static', 'guided', 'dynamic'). This can be set at runtime with the environment variable 'OMP_SCHEDUlE'. The performance of 'seidel is improved with 'OMP_SCHEDULE' set to 'guided' with 'PLUTO'. But for 'adi' even without help from PLUTO, just setting 'OMP_SCHEDULE' was enough to improve performance. The results are shown in Table 5.2

	Serial Execution	**Polly + OpenMP**	**Polly + PLUTO + OpenMP**
2 Core 32 Bit	0.417174s	0.591673s	0.348909s
2 Core 64 Bit	0.310160s	0.459641s	0.254605s

Table 5.2: Performance improvement of seidel

Another interesting result observed is when 'seidel is tested in 10-core machine with larger data size(N=4096). There is significant variations when the OpenMP parameters are tuned to different values. Optimizations is done with the combination of Polly and PLUTO. The scheduling policy is set to 'guided' and other parameters(chunk size and number of OpenMP threads) are varied. The results are show in Table 5.3

Chunk size / No of threads	**512**	**256**	**128**
default	12.930170s	11.254353s	37.003882s
10	15.433336s	14.657253s	14.518356s
5	14.002886s	12.283284s	14.018281s
2	16.649145s	18.778266s	18.013177s

Table 5.3: Performance of seidel with different OpenMP parameters

A sub project for Polly is already planned to increase the coverage and performance of Polly, which will consider various possibilities for improvement. For details refer to Chapter 6.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

Polly integrated with automatic OpenMP code generation helps users to realize parallelism without much worries about the internals of the program. Lot of effort and programming time can be saved because the approach eliminates the need for manually finding parallelism and providing annotation. Since the optimizations are performed on LLVM-IR the framework is not restricted to only C/C++, but also supports a wide range of other programming languages. Powerful polyhedral techniques are available to optimize programs, but their impact is still limited. The main reasons are difficulties to optimize existing programs automatically and generate parallel code. Polly and its integrated OpenMP support is an attempt to strengthen the impact. There is enough space for further enhancements and anybody interested can make their contribution. There are lot of possibilities for improving Polly. Some of the subprojects that can be done are discussed here.

6.2 Support for memory access transformations in Polly

An improvement that can be made to polly is to add support for memory access transformations in Polly. In many cases it would be great to change the pattern of

memory access to obtain better data locality. This can remove dependences that would otherwise block transformations and it can allow LLVM to use registers to store such values.

Polly performs its optimization on LLVM-IR based on the polyhedral model. Currently the transformations can be applied on Schedule (Order of computations) Transformations can also be applied on the Memory Access (Pattern of memory access). A proper memory access transformation can improve data locality. This will in turn improve parallelism.

6.3 Increasing coverage of Polly

Polly (Polyhedral optimization framework in LLVM) is showing very nice results for several testcases. Yet, lot of larger test cases needs to be improved. we can explore the reasons for this, find solutions for those and implement it. There are two parts for this.

6.3.1 Increasing SCoP coverage

The number of SCoPs detected need to be improved. This can be called as ”Increasing SCoP Coverage”.

Expressions like min, max, sext, zext, trunc or unsigned comparisons in the loop bounds or memory accesses are not handled in the current implementation. For example consider the following loop.

```
for (int i = 0; i < N; i++)
  A[i] = B[i] + C[i];
```

In this case a sext is necessary if the code is translated to LLVM-IR and keep i as an i32 and use an i64 to calculate the access to A[i]. This is not currently handled in Polly.

Overflows NSW(No signed wrap) or NUW(No unsigned wrap) are not handled in the current implementation. So it is not safe to compile a large project with Polly.

Polly can be tested with large benchmarks like SPEC and there is a very high possibility for finding areas which are not detected as SCoPs. It will be interesting to explore the reasons for this fix it.

6.3.2 Increasing the system coverage

Some of the testcases are failing when Polly is tested in machines which does not have 64-bit Operating system. This needs to be fixed and can be called as "Increasing the System Coverage". This can also be treated as porting to Polly to more architectures. A solution for this issue could be to derive the data type needed by the maximal possible value a variable can have.

6.4 Integrating profile guided optimization into Polly

An improvement that can be made to Polly is integrating profile guided optimization [6]. The idea is explained below with a few examples. Consider the following code.

```
scanf("%d", &b);
for(i = 0; i < N; i += b) {
    body;
```

```
}
```

Polly will not detect this as a SCoP because the variable b is read as a user input. So to detect this as a SCoP we instrument the IR with the information provided by profiling. Suppose using profiling we figure out that most of the time the value of b is say 2. we can convert the above code as follows.

```
scanf("%d", &b);
if(b == 2) {
  for(i = 0; i < N; i += 2) {
      body;
  }
} else {
    for(i = 0; i < N; i += b) {
        body;
    }
}
```

Now with the transformed code the for loop inside 'if' will be detected as a SCoP and can be parallelised. Since value of b is 2 most of the time, the overall performance will be improved.

Consider another scenario.

```
  for(i = 0; i < N; i++) {
      body;
  }
```

Suppose using profiling we know that N is always very small. So there will not be much gain from parallelising it. So we have to tell polly that do not detect this as a SCoP if N is less than a specific value. Integrating such versioning we can expect to get heavily optimized performance for some often used cases.

APPENDIX A

Setting up the environment

The source code for Polly can be downloaded from http://repo.or.cz/w/polly.git. This section describes how to set up the environment to work with Polly. We have to install libgmp, llvm-gcc/clang executables, cmake, llvm, ClooG, Pocc and scoplib. libgmp, llvm-gcc/clang and cmake can be installed using the package management system of the operating system. Installing others are explained below

A.1 CLooG

Get the source code of ClooG using the command

```
git clone git://repo.or.cz/cloog.git
```

Get the required submodules and build it

```
cd cloog
./get_submodules.sh
./autogen.sh
./configure --with-gmp-prefix=/path/to/gmp/installation \
                    --prefix=/path/to/cloog/installation
make
make install
```

A.2 PoCC

Get a released source code of PoCC with

```
wget http://www.cse.ohio-state.edu/~pouchet/software/ \
        pocc/download/pocc-1.0-rc3.1-full.tar.gz
```

Extract the tarball and build

```
tar xzf pocc-1.0-rc3.1-full.tar.gz
cd pocc-1.0-rc3.1
./install.sh
export PATH=`pwd`/bin
```

A.3 Scoplib

Get a released source code of Scoplib with

```
wget http://www.cse.ohio-state.edu/~pouchet/software/ \
        pocc/download/modules/scoplib-0.2.0.tar.gz
```

Extract the tarball and build

```
tar xzf  scoplib-0.2.0.tar.gz
cd scoplib-0.2.0
./configure --enable-mp-version \
        --prefix=/path/to/scoplib/installation
make && make install
```

A.4 Building LLVM with Polly

Download the source code of LLVM

```
git clone http://llvm.org/git/llvm.git
```

Download the Polly source into the 'tools' directory of LLVM and build it using 'cmake'

```
cd llvm/tools
git clone git://repo.or.cz/polly.git
mkdir build
cd build
cmake <llvm_source_dir>
# If CMAKE cannot find CLooG and ISL
cmake -DCMAKE_PREFIX_PATH=/path/to/cloog/installation .
# To point CMAKE to the scoplib source
cmake -DCMAKE_PREFIX_PATH=/path/to/scoplib/installation .
make
```

APPENDIX B

Various Tools Used in Polyhedral Community

Some of the tools used in polyhedral community is described here with one example.

B.1 ClooG

CLooG is a free software and library generating loops for scanning Z-polyhedra. It finds a code that reaches each integral point of one or more parameterized polyhedra. CLooG has been written to solve the code generation problem for optimizing compilers based on the polytope model.

This is explained here with a simple example. Suppose we need to generate code for a polyhedra with the following iteration domain.

$$D_S = \{(i, j)\, \epsilon\, Z^2 \mid 2 \leq i \leq 6 \wedge 2 \leq j \leq 6 \wedge i \leq j\}$$

We can give input to cloog as file which has the following format.

```
# ---------------------- CONTEXT -------------------
c # language is C
# Context (constraints on two parameters)
2 4                    # 2 lines and 4 columns
# eq/in m  n  1    eq/in: 1 for >=0, 0 for =0
0   1  0 -6         # 1*m + 0*n -6*1 >= 0, i.e. m=6
```

```
0    0   1  -6        # 0*m + 1*n  -6*1 >= 0, i.e. n=6
1 # We want to set manually the parameter names
m n                    # parameter names
# ---------------------- STATEMENTS -----------------
1 # Number of statements
1 # First statement: one domain
# First domain
5 6                    # 5 lines and 6 columns
# eq/in i   j   m   n   1
1    1   0   0   0  -2 # i >= 2
1   -1   0   0   1   0 # i <= n
1    0   1   0   0  -2 # j >= 2
1    0  -1   1   0   0 # j <= m
1   -1   1   0   0   0 # n+2-i>=j
0   0   0              # for future options
1 # We want to set manually the iterator names
i j                    # iterator names
# ---------------------- SCATTERING ----------------
0 # No scattering functions
```

Giving this file to 'cloog' command as input it will generate the following code.

```
/* Generated from ex1.cloog by CLooG 0.16.2 gmp bits in 0.00s. */
for (i=2;i<=6;i++) {
  for (j=i;j<=6;j++) {
    S1(i,j);
  }
}
```

B.2 PLUTO

PLUTO is an automatic parallelization tool based on the polyhedral model. The polyhedral model for compiler optimization is a representation for programs that makes it convenient to perform high-level transformations such as loop nest optimizations and loop parallelization. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling and fusion, but not limited to those. Suppose we want to generate the tiled version of lu.c just issue the command ./polycc –tile lu.c which will generate the output in a separate file called lu.tiled.c and displays the following information

```
[Pluto] Number of statements: 2
[Pluto] Total number of loops: 5
[Pluto] Number of deps: 10
[Pluto] Maximum domain dimensionality: 3
[Pluto] Number of parameters: 1
[Pluto] Affine transformations [<iter coeff's> <const>]
T(S1): (k, j, k)
3 3
 1  0  0
 0  1  0
 1  0  0
T(S2): (k, j, i)
3 4
 1  0  0  0
 0  0  1  0
 0  1  0  0
t1 --> fwd_dep  loop   (band 0)
t2 --> fwd_dep  loop   (band 0)
t3 --> fwd_dep  loop   (band 0)
```

```
[Pluto] Outermost tilable band: t0--t2
[Pluto] After tiling:
t1 --> fwd_dep  tLoop  (band 0)
t2 --> fwd_dep  tLoop  (band 0)
t3 --> fwd_dep  tLoop  (band 0)
t4 --> fwd_dep  loop   (band 0)
t5 --> fwd_dep  loop   (band 0)
t6 --> fwd_dep  loop   (band 0)
[Pluto] using Cloog -f/-l options: 4 6
[Polycc] Output written to ./lu.tiled.c
```

B.3 VisualPolylib

This is a tool using which we can visualize various operations on polyhedra. The polyhedral description is given in a file and given as input to the 'visualpolylib' command. For instance consider the following input

```
P :=   { i, j, k | k <= 1, k >= 0, i <= j, i+j >= k, i+4k <= 4j,j<=10};
C:={|};
CS := { i,j,k | i-2j+2k<=0};
P2 := CS . P;
initvisu(3,C)
```

The polyhedron for P2 which is the intersection of CS and P can be viewed graphically as in Figure B.1

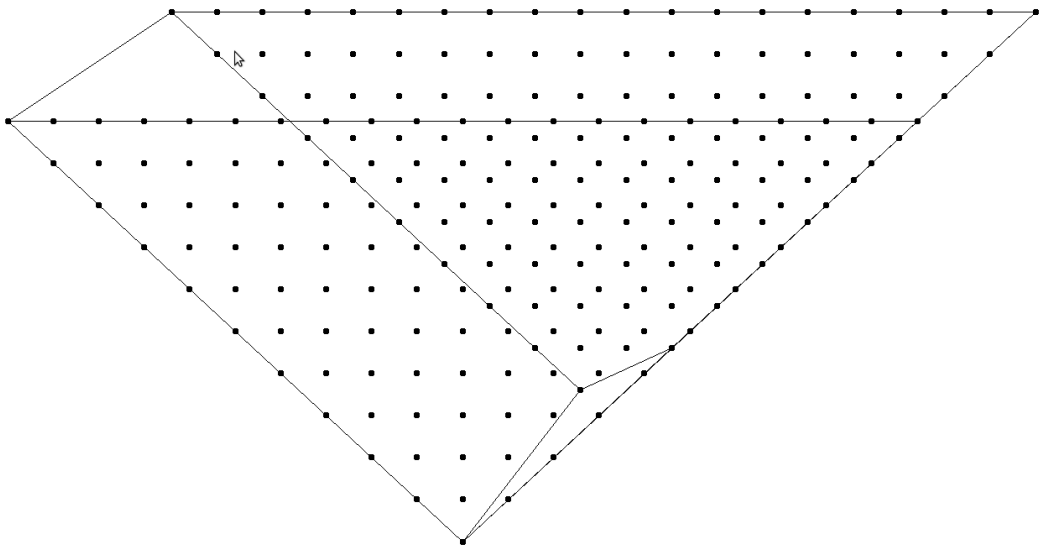


Figure B.1: Visualizing polyhedra with VisualPolylib

Publications

1. Tobias Grosser, Hongbin Zheng, **Raghesh Aloor**, Andreas Simbürger, Armin Größlinger and Louis-Noël Pouchet Polly - Polyhedral optimization in LLVM *IMPACT 2011(First International workshop on PolyhedrAl Compilation Techniques as part of CGO 2011)*, Chamonix, France.

REFERENCES

[1] **Bastoul, C.**, Code generation in the polyhedral model is easier than you think. *In PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 2004. URL `http://hal.ccsd.cnrs.fr/ccsd-00017260`. Classement CORE : A, nombre de papiers acceptés : 23, soumis : 122, student award.

[2] **Benabderrahmane, M.-W.**, **C. Bastoul**, **L.-N. Pouchet**, and **A. Cohen** (2009). A conservative approach to handle full functions in the polyhedral model. Technical Report 6814, INRIA Research Report.

[3] **Bondhugula, U.**, **A. Hartono**, **J. Ramanujam**, and **P. Sadayappan**, A practical automatic polyhedral parallelizer and locality optimizer. *In Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-860-2. URL `http://doi.acm.org/10.1145/1375581.1375595`.

[4] **Crockford, D.** (2006). The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational). URL `http://www.ietf.org/rfc/rfc4627.txt`.

[5] **Girbal, S.**, **N. Vasilache**, **C. Bastoul**, **A. Cohen**, **D. Parello**, **M. Sigler**, and **O. Temam** (2006). Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, **34**, 261–317.

[6] **Gupta, R.**, **E. Mehofer**, and **Y. Zhan** (2002). Profile guided compiler optimization. *The Compiler Design Handbook: Optimizations and Machine Code Generation*.

[7] **Johnson, R.**, **D. Pearson**, and **K. Pingali**, The program structure tree: computing control regions in linear time. *In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94. 1994.

[8] **Lattner, C.** and **V. Adve**, Llvm: A compilation framework for lifelong program analysis & transformation. *In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2102-9. URL `http://portal.acm.org/citation.cfm?id=977395.977673`.

[9] **Trifunovic, K.**, **A. Cohen**, **D. Edelsohn**, **F. Li**, **T. Grosser**, **H. Jagasia**, **R. Ladelsky**, **S. Pop**, **J. Sjödin**, and **R. Upadrasta**, GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. *In GCC Research Opportunities Workshop (GROW'10)*. Pisa Italy, 2010.

[10] **Verdoolaege, S.**, Isl: An integer set library for the polyhedral model. *In Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*. 2010, 299–302.