POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Detecting Data Access Patterns in OpenMP Parallel Loops

Formal Languages and Compiler Group
Politecnico di Milano

Relatore: Ing. Giovanni Agosta
Correlatore: Ing. Ettore Speziale

Tesi di Laurea di:
Maggioni Marcello, matricola 754531

Anno Accademico 2011-2012

Abstract

The increasing difficulty in improving single thread performance has led to a major shift towards multi-processor and multi-core systems. Shared memory architectures provide the best programmability features for multi-processor systems, due to the simple abstraction provided by the shared memory. Due to the technology constraints in the scalability of memory access, *Non Uniform Memory Access* (NUMA) architectures have been proposed. NUMA architectures aim at reducing memory bus stalls by providing a separate bus from each processor to a portion of the memory (”local” to that processor). Access to that portion of the memory from other (”remote”) processors is guaranteed, but remote processors need to pass through the local processor, leading to an increased access latency. The locality of memory access, therefore, becomes pivotal to performance.Thread schedulers unaware of data locality assign threads to processors randomly, leading to sub-optimal solutions. To improve over this baseline, it is necessary to take into consideration which memory addresses are accessed by each iteration of a loop — the *data access pattern* of that loop. With this knowledge, it is possible to select the best NUMA node for the execution of each loop iteration, and schedule the threads accordingly.In state of the art solutions, the data access pattern information is encoded manually by the programmer by means of extensions of the OpenMP `parallel for` directive. The goal of this work is to automatically derive the necessary information, speeding up the development of parallel code and reducing the need for the developer to have a deep understanding of the performance of the program.

To this end, we have developed compiler analysis and transformation passes that compute the data access pattern information and provide it to the lower stages of the compilation process. The development of this pass is the objective of this Thesis.

Abstract

La difficolta' crescente nell' aumentare le performance single-thread ha portato allo sviluppo sempre crescente dei sistemi multi-processore e multi-core. Le architetture con memoria condivisa offrono la migliore programmabilita' sui sistemi multi-processore, grazie alla uniformita' dello spazio di memoria. A causa di difficolta' di scalabilita' dei sistemi con molti nodi nell'accesso di memoria, sono state sviluppate le architetture *Non Uniform Memory Access* (NUMA). Le architetture NUMA puntano a ridurre gli stalli del bus di memoria offrendo un bus separato, per ogni processore, a una porzione della memoria ("locale" a quel processore). L'accesso a quella porzione di memoria da altri ("remoti") processori e' garantita , ma i processori remoti devono passare attraverso il processore locale, portando ad una latenza incrementata. La localita' dell' accesso di memoria, quindi, diventa un importante fattore per le performance delle applicazioni su questi sistemi. Gli scheduler dei thread, non a conoscenza di informazioni riguardo la localita' della memoria acceduta nel thread, assegnano i thread ai processori spesso in maniera non ottimale. Per migliorare la situazione e' necessario conoscere quali indirizzi di memoria sono acceduti da ogni iterazione di un loop - ossia lo *schema di accesso alla memoria* di quel loop. Con questa conoscenza, e' possibile selezionare il miglior nodo NUMA per l'esecuzione di ogni iterazione assegnando i thread in maniera ideale. Le attuali soluzioni allo stato dell'arte permettono di specificare manualmente lo schema di accesso della memoria dei thread attraverso estensioni alla sintassi della direttiva OpenMP **parallel for**. L'obiettivo di questo lavoro e' di riuscire a derivare le informazioni necessarie automaticamente, facilitando lo sviluppo di codice parallelo e riducendo la necessita' per una profonda conoscenza delle interazioni con la memoria da parte del programmatore.

A questo scopo, e' stata sviluppata una serie di passi di analisi e trasformazione del codice che calcola lo schema di accesso dei dati ed e' in grado di fornirla agli stadi successivi del processo di compilazione o direttamente al programmatore. Lo sviluppo di questi passi e' l'obiettivo di questa tesi.

Contents

List of Figures

Nomenclature

API	Application Programming Interface
ccNUMA	Cache-coherent Non-Uniform Memory Architecture
CFG	Control-flow graph
GEP	GetElementPtr
ISL	Integer Set Library
LLVM	Low-Level Virtual Machine
LLVM-IR	LLVM Intermediate Representation
NUMA	Non Uniform Memory Access
OpenCL	Open Computing Language
OpenMP	Open Multiprocessing
SCEV	Scalar Evolution
SCoP	Static Control Part

Part I

Introduction

Chapter 1

Motivation

Moore's law [1] has always been a driving force for microprocessor development. This law has been followed very closely in the past years by adding, to each new generation of microprocessors, components like branch predictors , larger caches, additional scalar and vector execution units. All these new components brought massive performance increases in the single threading domain, but in the recent years a wall has been hit because of difficulties found in rising operating frequencies of high performance microprocessors and new extensions to these already mentioned components aren't enough to reach the expected increase of performance. Now the development shifted toward multicore/multiprocessor systems. These systems try to deliver increased performance by using multiple processors on the same system or adding more processors in the same chip package. This approach works well , but may be problematic, in particular because of two problems:

- Multithreading programming isn't easy to do and more often than not programs are not optimized for multithreading systems by having big single threaded parts that limit the performance in the terms of the Amdahl's law. [2]
- Symmetric Multicore/Multiprocessor systems scale very well with small numbers of processors, but start to lag behind when a lot of processors access the same bus

1.1 Issues in Parallel Program Development

Parallelizing the code of a program is not an easy task. Identifying different components to parallelize is complex, threading APIs aren't straight forward and, usually, aren't cross platform (because they are strictly tied to the OS), in addition the interaction between parallel sections of a program may bring up race conditions, deadlocks and other typical problems of parallel code. Because of this there has been work by software engineers in the recent years towards making it easier for programmers to produce parallel code by developing easier programming APIs and tools. Like the first compilers helped early programmers in making serial programming easier thanks to the employment of high-level programming languages instead of machine languages, these APIs aim to let the programmers concentrating more on the solution they have to develop and less on the lower level matters of parallelization. Examples of these new tools are **OpenMP** (Task parallelism) and **OpenCL** (Data and Task Parallelism). In particular OpenMP is specific for CPU program parallelization while OpenCL is more focused on heavily parallelized devices, like GPUs (but many implementations also support CPU devices). These provide tools to define parallel Task and Data domains.

Loop Parallelization OpenMP has an interesting feature that is loop parallelization. This feature makes it possible to schedule loop iterations over multiple threads with just a line of code and needs little human intervention (that is what makes this feature so interesting). Loops with many data independent iterations can get very big speedups thanks to this.
Loop parallelization is explained later in greater detail in Chapter 5

1.2 Beyond the limit of SMP systems

Faster buses and larger caches have been used to avoid performance problems related to memory bus accesses as much as possible. After a certain amount of nodes these solutions are not enough. The solution that is used in big multiprocessor systems is called **ccNUMA** . In NUMA systems, the memory address space is shared with all the other processors, like in SMP, but the difference with SMP is that memory is not directly connected to all the nodes through a common bus, but each NUMA node (that can be composed of one or more processors) have only a certain amount of memory directly connected to their local bus. The memory connected to the other nodes is accessed through an interconnect bus that is much slower. The whole memory address space is available in both SMP and NUMA systems, but in the latter the access cost of different parts of the address space is different and not uniform (hence the name).

NUMA helps in reducing the bus race conditions that usually happen in large SMP systems by partitioning it into smaller nodes.

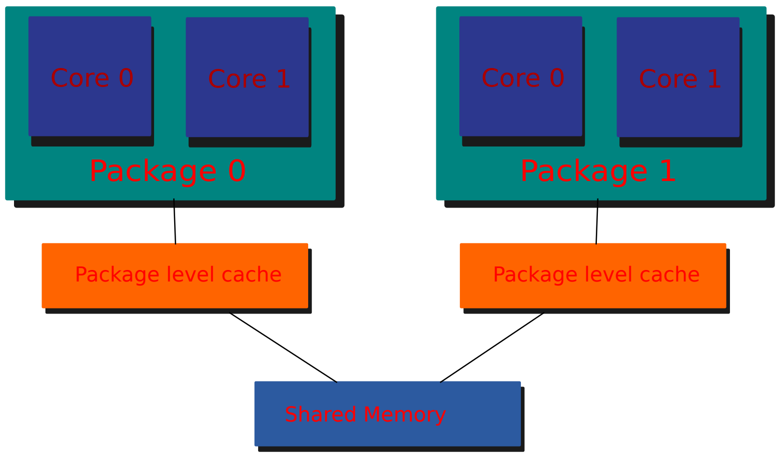


Figure 1.1: A typical SMP system

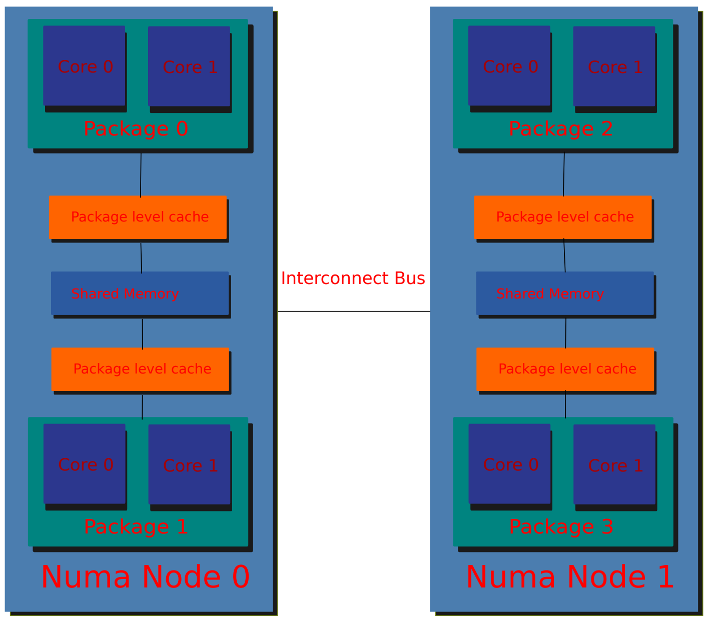


Figure 1.2: A typical NUMA system

1.3 Parallelization for NUMA systems

For the best performance results NUMA programs should be hand-tuned in order to leverage the better performance of local memory compared to the one connected to other NUMA nodes. This requires deep knowledge of the system and of the access patterns to memory of the various threads composing the program. When loop parallelization is used on NUMA systems the generated code might not be optimal, because of the uninformed and automatic nature of this approach, which usually produces inefficient code for these systems. Some steps to solve this have already been done. Some researchers at Politecnico di Milano developed a modified OpenMP runtime that is able to accept additional information, fed by the programmer, about the memory access patterns inside OpenMP loops and to schedule the loop threads to the best system node.[9] The problem of this approach is that it would be better if the programmer isn't burdened with the task of determining the memory access patterns of the program, because of the inherent low-level nature of the task and the effort it takes.

1.4 Objective of the Thesis

The final objective of this Thesis is to find a way to expand on the previous work on this subject and remove this requirement by producing a compiler analysis pass that determines the memory access patterns of the program and produces the the meta-data that the programmer had to determine by himself using polyhedral analysis of the parallelized OpenMP loops.

1.5 Structure of the Thesis

In the next chapters will be given an introduction to the tools and the concepts used to reach the stated objective. The introduction will help in better understanding the choices made in developing the solution. After this the solution itself and the work produced will be presented in detail.

Part II

Tools

Chapter 2

OpenMP

The work in this Thesis targets OpenMP. OpenMP is an API that provides to programmers a portable , shared-memory model interface to build parallel applications for a broad range of devices, from desktops to supercomputers. The OpenMP specification is publicly available on the OpenMP website [1] and is maintained by an Architecture Review Board (ARB) composed of some of the most important players in the computing field , like Microsoft, IBM, Intel and AMD. The ARB is responsible for the future developments of the OpenMP specification. OpenMP uses the traditional operating system threading API to parallelize the program, so it can be thought as a portable and simplified interface to that same API.

2.1 Execution Model

The **execution model** of OpenMP is based on a model called **fork-join**[3]. When an OpenMP program starts it spawns a thread like any other program: that thread is called the **Master Thread** and it is the main thread of the program. An OpenMP C++ program is instrumented through the use of special preprocessor pragma directives that the compiler uses to determine how to handle and parallelize the code. The most important of all these directives is the **parallel** directive. When a parallel directive is reached OpenMP spawns new threads (the default number of threads is implementation dependent and can be specified by the user). The code contained in the code block next to the parallel directive is executed by all the threads concurrently. At the end of the code block a barrier joins all the spawned threads together and then the execution continues with only the master thread surviving the join event. It is easy in this way to spawn new threads and execute sections of code

[1]http://www.openmp.org

concurrently.

```
int main()
{
  #define omp parallel num_threads(2)
  {
    printf("Multithread Hello World!");
  }
  return 0;
}
```

Figure 2.1: OpenMP parallel directive. Here the text is displayed twice.

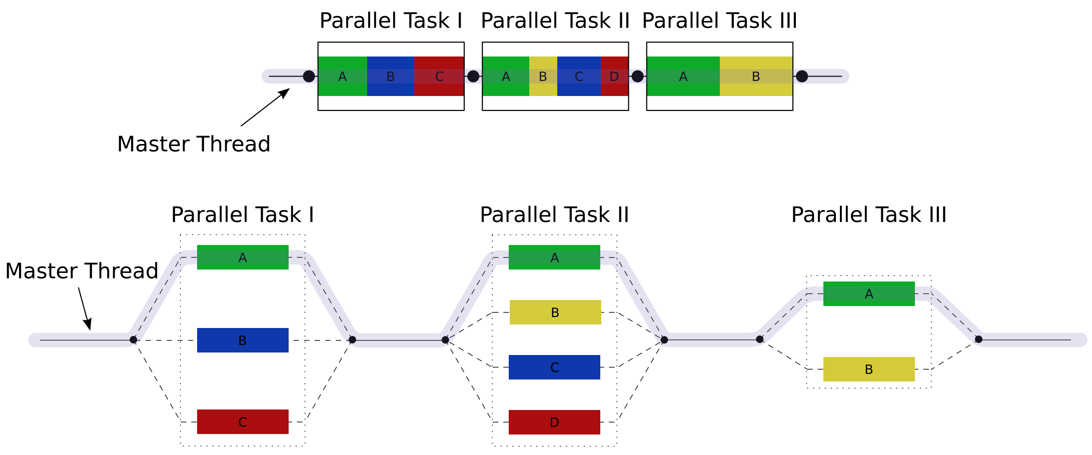


Figure 2.2: Fork-Join model

Inside a **parallel** region many different OpenMP directives can be inserted. Each of these statements have a different meaning and interact with the currently available threads spawned by the the **parallel** directive distributing work between them or synchronizing them. An example of these additional directives are the **sections** and **section** directives that can be used to define code blocks that have to be executed by only one thread of the group of threads spawned by the parallel directive so that the code blocks defined can be distributed between the worker threads.

```
int main()
{
  #pragma omp parallel num_threads(2)
  {
    #pragma omp sections
    {
      #pragma omp section
      {
          printf ("id = %d, \n", omp_get_thread_num());
      }

      #pragma omp section
      {
         printf ("id = %d, \n", omp_get_thread_num());
      }
    }
  }
  return 0;
}
```

Figure 2.3: Example of usage of the sections directive. Each thread prints it's own id number.

The OpenMP execution model can so be summarized in this way:

1. Define a parallel section that spawns a new team of threads to be used for task execution

2. Write code that needs to be run multiple times (in multiple threads) or specify other OpenMP directives to distribute the work between the threads in the team.

3. If needed synchronize between the threads using the **barrier** OpenMP directive or other synchronization OpenMP directives.

2.2 Memory Model

The **memory model** is how OpenMP manages memory and how the various threads share it. The OpenMP memory model is quite simple and is very similar to the traditional threading model found in the most famous operating systems, where the memory is shared between the threads but it is possible for threads also to have private memory. It is possible to specify which variables must be private at the definition of specific OpenMP directives, in which case each reference inside the block for the directive will refer to a private copy of the variable created for each of the threads. Accessing to shared memory doesn't necessarily mean that memory is immediately updated, but OpenMP uses a **relaxed-consistency** memory model that gives to the implementations the freedom to create a local memory view even for shared memory variables accesses that are periodically synchronized with the shared memory. This mechanism is transparent to the user and can't be controlled, with the exception of a specific OpenMP clause , the **flush** clause, that can be used to synchronize the global memory with the local view.

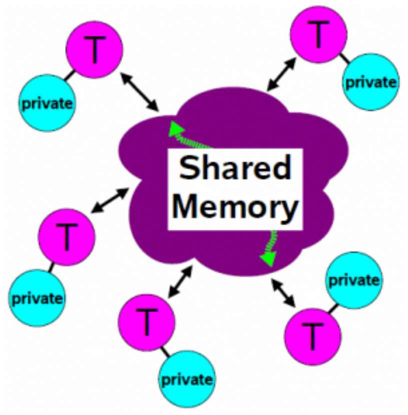


Figure 2.4: Memory model

2.3 Loop Parallelization

One of the most performance critical parts of program code are loops. Loops are parts of code executed multiple times, so they are the perfect target for optimization, and where usually performance hogs are found. Of the many possible ways to increase loop performance, **Loop Parallelization** is one of particular interest thanks to the recent widespread adoption of multi-core architectures.[4] In loop parallelization the various loop iterations are distributed to different threads for execution. While there are some compilers that execute this optimization automatically the optimization is still usually hand-made. The reason is that compilers have to do memory dependence analysis on the loop iterations in order to determine which iterations can be parallelized and which must be executed serially. This is a complex task to do and not all optimization opportunities are exploited from compilers. **OpenMP** helps the programmer in doing this optimization manually by providing syntactical facilities.

Here is provided an example of usage of OpenMP to define parallel loops:

```
void simple(int n, float *a, float *b)
{
    int i;
#pragma omp parallel for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Figure 2.5: OpenMP parallel loop example. Each loop iteration is dispatched to a thread (the number of them may vary)

This code snippet produces a for loop whose iterations are distributed among the spawned threads by the **parallel** directive. In this case the **parallel** and the **for** directives (which is the OpenMP loop directive for C/C++ code) have been fused together in one single ”parallel for” directive. OpenMP permits this kind of fusion between parallel and other statements that are often the only ones present inside a single parallel block.

Chapter 3

The Polyhedral Model

The Polyhedral Model is a way of representing and manipulate loop structures and statements. The iteration space is modeled through a polytope.[7] A huge number of loop transformations can be implemented using the polyhedral model , like loop fusion, loop fission, strip-mining and data dependence analysis. In the Polyhedral Model each statement of the loop is represented as one point inside the bounds of a polytope for each time it is executed. The bounds of the polytope are the representation of the loop bounds. Not all loops are suitable to polyhedral analysis, because they need to have some specific features to make them compatible with this kind of analysis, in particular:

1. The loop bounds must be defined (so while loops usually are not suitable to polyhedral analysis)
2. The loop bounds must be affine functions of the loop iterators
3. Conditions contained in the loop must be affine
4. Loop iterators can't be passed as modifiable parameters (through pointers or references) to a function
5. Loops must have a constant stride

If a loop has these properties then it can be represented through the polyhedral model and can be part of a **SCoP** (Static Control Part).

3.1 SCoPs

A SCoP is the maximal set of consecutive statements without while loops where loop bounds and conditionals may only depend on invariants within this set of statements[10]. These invariants include symbolic constants, formal function parameters and surrounding loop counters. These are called *global parameters* of the SCoP.

```
int i;
for (i=0; i < n; i++) {
//-- SCoP 1 begins here -- three statements, Iterators: i,j Parameters: m,n,k
  S1
  int j;
  for (j = 0; j < m; j++) {
    S2
    if ( j < k )
      S3
  }
}
```

Figure 3.1: An example of a SCoPs

The SCoP in the example above defines two polytopes. An one dimensional polytope for S1 and a two dimensional polytope of rectangular shape for S2 and S3. Each point inside the polytope defines an iteration of the loop modeled by the SCoP. A statement may or may not execute when a specific iteration of the loop is executed. In Figure 3.2 are represented statement instances for statements S2 and S3. A point is present in the graph when a certain statement is executed at loop iterators values **i** and **j**.

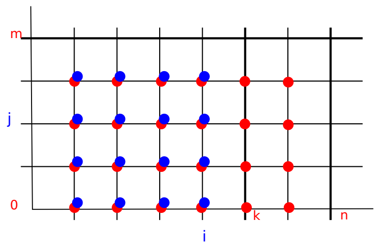


Figure 3.2: Statement instances for S2 (red) and S3 (blue) in the SCoP

The values for which a SCoP statement is executed are called the **domain** of the statement. For example for statement S1 the domain is $\{i|i \geq 0, i < n\}$ while for S2 the domain is $\{i, j|i \geq 0, i < n, j \geq 0, j < m\}$ and for S3 is $\{i, j|i \geq 0, i < n, j \geq 0, j < m, j < k\}$.

3.1.1 Scattering

The statements are also characterized by the order in which they execute, with respect to each other, that is called **scattering**. The scattering determines the scheduling of execution of statements. Statements with lower scattering values are executed before statements with higher values. The scattering is usually represented through vectors of multiple elements like $S_2s = (0, i, 1, j, 0)$, which can be a representation of the scattering for statement number 2. For statement number 3 a possible scattering would be $S_3s = (0, i, 2, j, 0)$. The S3 vector has an higher lexicographical value than that of S2, signifying that statement 3 is executed after statement 2. The amount of elements of the scattering vector is usually $2 \star NumberOfLoopsOfTheSCoP + 1$, but the format of the scattering vector may be arbitrary and is not standardized. At last a valid scattering for statement number 1 could be $S_1s = (0, i, 0, 0, 0)$. These scattering vectors define the correct execution order for the statements in the SCoP. Changing values contained in the scattering vectors changes the execution order of the various statements and it is what transformation passes actually do when the aim of the transformation is changing the execution order of statements in the loop.

3.1.2 Memory Accesses

Considering that reordering statements may also change the order of memory accesses (because usually statements access memory in some way, either by reading or writing it) these are important components that define statements. Memory accesses may be dependent from one another and because of this usually transformations that involve swapping the statements execution order must take in consideration memory dependencies to make transformations that do not change the semantics of the program. A statement may do zero, one or many memory accesses. Of all the memory accesses the best ones for the polyhedral model are those with a pattern that can be represented through an affine function (with respect to the loop iterators). Affine memory accesses can be easily analyzed and the dependency between accesses can be derived.

```
int i,j, A[100][100];
for (i=0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
    A[i][j] = i;
  }
}
```

Figure 3.3: Example of an affine memory access

In the example of Figure 3.3 memory is accessed with an affine access function. The access function is $A(i, j) = 100 \star j + i + BASE$ where $BASE$ is the base pointer of the data structure being accessed, in this case the address of the first element of array A. Example of non-affine memory accesses are those using a non-affine access function of the loop iterators. Function calls may also be considered potential non affine memory accesses, because they may make arbitrary memory accesses that can't be known in advance (the code for the function itself may not be available at compile time, like for example those in external dynamically-linked libraries).

Chapter 4

LLVM and Polly

The **LLVM** compiler infrastructure and it's polyhedral analysis tool called **Polly** are the main tools used in this Thesis. LLVM is a set of libraries developed specifically for compiler development, written in C++ and originally developed at the University of Illinois. [6] Later, the core development has been taken over by Apple that uses it as the main compiler technology for their systems like the MacOS and iOS.[5] It is an Open Source project open to anyone to contribute under a BSD-like license and actively developed. LLVM has not been created as a complete compiler itself, but it provides the basis to build front-ends, optimizers and code generators both for static and dynamic compilation.

4.1 High level structure of LLVM

LLVM is a set of libraries aimed at compile-time, link-time and run-time optimization of code.[8] In order to do that in a machine independent way LLVM components interact with the code (almost) completely when it is converted into its internal LLVM-IR (LLVM Intermediate Representation). The LLVM-IR is a RISC-like register-based assembly language used as an intermediate representation for the LLVM optimization passes. Program code in LLVM-IR can be constructed through a specific API that is part of LLVM. After the code is generated it can be altered through the various LLVM optimization passes. LLVM provides interfaces to build different kinds of optimization passes depending on the level it has to work on. For example if the optimization has to work at the Loop level then a LoopPass can be built, if instead it has to consider the compilation unit as a whole then a ModulePass can be used. After the code has been analyzed and altered by the various optimization passes then the code can be translated to native code through the native code generators. The generated code can either be stored on the hard-drive and linked to form a static native executable or it can be directly executed (this enables the development of JITs through LLVM).

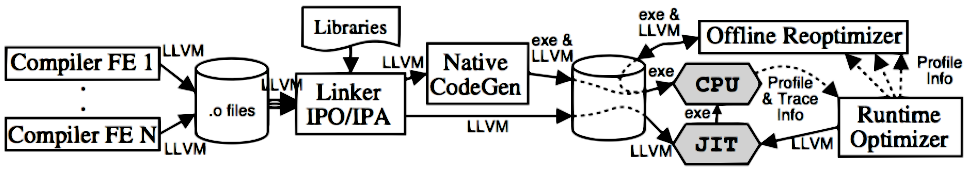


Figure 4.1: High level LLVM structure

The overall structure of a compiler built on LLVM is then:

1. A Front-End that parses and translates the original source code to LLVM-IR code using the LLVM API to construct the basic-blocks.
2. Optimization passes to be run on the generated IR code.
3. The native code generator that outputs the native code eventually applying native code optimizations to the generated code.

This project is itself a collection of LLVM optimization passes (both analysis and transformation passes for code normalization).

4.2 Polly

Polly is a polyhedral analysis tool for LLVM.[11] It is a collection of LLVM transformation and analysis passes that find the valid SCoPs inside the program, construct their representation (statements domain, scattering,memory accesses and dependencies), apply transformations to the code and regenerate the code from the modified polyhedra to obtain the optimized code from the representation. Optimizations based on the polyhedral model can be built on top of Polly by writing specific Polly passes that are run once for every valid SCoP detected. This project uses Polly as a base for analyzing the memory accesses inside loops and determining, from their polyhedral representation, the access patterns of memory access.

4.2.1 Polly internal representation and ISL

When Polly analyzes a SCoP it extracts all the relevant information of it like the global parameters, the statements domain, scattering and memory accesses and stores them in internal data structures using ISL objects (Integer Set Library). ISL is a C library for manipulating sets and relations of integer points bounded by affine constraints.[12] The library supports a wide range of operations on integer sets and relations and operations on affine functions (like addition, subtraction, multiplication). ISL is used by this project to do actual computations on affine memory access functions using the data collected by Polly.

Part III

Solution

Chapter 5

Data Access Pattern Detection

5.1 OpenMP Loop parallelization

When OpenMP parallelizes a loop it basically assigns the execution of the loop body to a thread spawned by an OpenMP parallel directive. A special function is automatically generated by the compiler containing the the code of the loop and used as the thread run function. If there are nested loops only the most external of the loops is parallelized. The nested loops are executed serially as part of a single iteration of the external loop.

```
int i,j, A[100][100];
#pragma omp parallel for schedule(dynamic)
for (i=0; i < 100; i++) {
  for (j = 0; j < 100; j++) {
    A[i][j] = i;
  }
}
```

Figure 5.1: Example of nested OpenMP loops. Only the red highlighted for is parallelized

The main consequence of this is that the interesting part is only the memory access pattern of the most external loop of the nest. In particular the objective is to find the range of memory addresses that each iteration of the external loop visits in some way (either by reading or writing it). This information can then be supplied to the OpenMP runtime that can then produce an adequate schedule.

OpenMP supports different scheduling schemes for dispatching loop iterations to the threads. The only scheduling scheme that will be supported is the **dynamic** scheduling (that is enabled by specifying the clause *schedule(dynamic)* at the definition of the OpenMP for directive). The dynamic scheduling approach distributes each loop iteration to the various threads as they are requested. To do so the compiler instruments the loop code adding function calls to the OpenMP runtime that inform it that the thread is ready to execute a loop iteration. The runtime returns to the thread if there are more iteration chunks to be executed and the values of the iteration counter for which to execute the loop code.

In Figure 5.2 is shown the translation of an example of a simple OpenMP dynamic for loop into the LLVM intermediate representation. The green basic block is the actual loop body, while the red basic blocks contains calls to a function called *GOMP_loop_dynamic_next()* that is an OpenMP runtime function used to obtain new loop iterations to execute. Depending on the value returned by this function the control flow executes the green basic block or the yellow basic block that instead contains a call to the *GOMP_loop_end_nowait()* function that informs the runtime that this thread finished its share of the loop workload and is exiting. Before passing control to the green basic block a red basic block is always executed to determine if the loop body has to be executed or the thread function has to terminate (by branching to the yellow block). The actual loop body can then be roughly defined as the sequence of basic blocks dominated by the two red ones.

5.2 Memory access patterns

The final objective is to extract information about the access patterns of multidimensional arrays of known dimensions. For multidimensional arrays each dimension can be accessed in different ways for each iteration of the OpenMP for loop. In Figure 5.3 an example of a possible access to a 2-dimensional array is presented. For each of the external loop iterations the nested loops access each element in *dimension 1* between the interval of elements 1 and 3 and this access pattern is replicated in direction of *dimension 2* two times. *Dimension 2* is accessed through the iterator of the most external loop. The result of this is that *Dimension 2* gets sliced horizontally. At the end of the execution of all the iterations of the most external loop the whole *Dimension 2* is covered. We can say that *dimension 2* is visited in slices of size 2. The final result here is that for each iteration of the external loop:

1. Array dimension 1 is accessed in the range of elements 1-3
2. Array dimension 2 is accessed in slices of size 2

A **Range** is then defined as the interval of elements accessed by each iteration of the most external loop for a particular array dimension. This array dimension is

not visited through the iterator of the most external loop.
A **Slice** is defined as the size of the chunk of elements visited by each iteration of the most external loop for a particular array dimension. This array dimension is visited through the iterator of the most external loop.

5.3 Finding the patterns

Polly provides data about the loops of the program being compiled in the form of SCoPs with all the information about the domain of the statements, their scattering and the memory accesses that each statement makes. Polly constructs a new statement for each basic block contained in the SCoP. All the instructions in a basic block are bound to be executed when the control flow passes through it, so all the instructions in a basic block share the same domain and scattering information.It is perfectly reasonable then to match basic blocks with SCoP statements in the polyhedral representation. For each memory access of a statement (in the form of a load or store instruction, that are the only LLVM instructions that access memory) Polly extracts the affine access function that represents the memory access.

5.3.1 SCoP loops bound detection

Using the data provided by Polly, the first step to extract pattern information is to find the **upper bounds** (UB) and **lower bounds** (LB) of each loop iterator for the statement we are currently analyzing. Because the SCoP is compatible with polyhedral analysis each loop contained in it must have definite lower and upper bounds. These don't need to be numerical constant bounds, but they must be some affine function of the SCoP global parameters. In Figure 5.4 is presented a simple example of a loop with a parametric upper bound and two statements. For loop iterator *"i"* the affine functions representing the upper and the lower bounds for statement 1 are $LB_i(\vartheta) = 0$ and $UB_i(\vartheta) = n - 1$ while for statement 2 the bounds are $LB_i(\vartheta) = 0$ and $UB_i(\vartheta) = 4 : n \leqslant 4$ or $UB_i(\vartheta) = n - 1 : n > 4$ where ϑ is the set of the global parameters of the SCoP. In the case of statement 2 we have two possible upper bound values depending on the value of the parameter *"n"*.

The iterator bounds can be derived from the domain of the SCoP statements. For statement 1 the domain for iterator *"i"* is $D_1 = \{i | i \geqslant 0, i < n\}$ while for statement 2 the domain is $D_2 = \{i | i \geqslant 0, i < n, i < 5\}$. The ISL library provides facilities to actually compute upper and lower bound of loop iterators given the domain of the statements (represented as n-dimensional integer sets in ISL. The number of dimensions of the set is the number of loop nests in the SCoP).

5.3.2 Access function computation

When Polly extracts memory access functions it linearizes them losing information about array dimensions. For example, if we consider the loop and the array in Figure 5.3 we can see that the array is multidimensional (2-dimensional to be exact) and for each array dimension we can identify an access function proper to that dimension. *Dimension 1* is accessed through the function $A_1(k) = k + 1$ while *dimension 2* is accessed through $A_2(i, j) = i + j$. What Polly does is linearizing the access transforming it into $A(i, j, k) = 5 \star i + 5 \star j + k + 1$. It is needed ,then, to get back each array dimension access function from the linearized one. This can be done by factorizing the linearized access function with the size of each array dimension. The algorithm to do this will be explained in greater detail later in the appropriate chapter.

5.3.3 Array dimensions bounds computation

After having derived the access functions for each array dimension the next step is to compute the lower and upper bounds of these access functions. The upper and lower bounds determine the range of values which the access functions span during each iteration of the most external loop. These functions have as arguments the iterators of the loop, substituting the upper and lower bounds of the loop iterators into these expressions returns respectively their upper and lower bounds. The iterator of the most external loop , though, must not be taken into consideration (it has to be treated as a constant). Going back to the example of figure 5.3 we have three loop iterators:

1. *"i"* that has an UB of 5 and an LB of 0 (this is the external loop iterator)
2. *"j"* that has an UB of 1 and an LB of 0
3. *"k"* that has an UB of 2 and an LB of 0

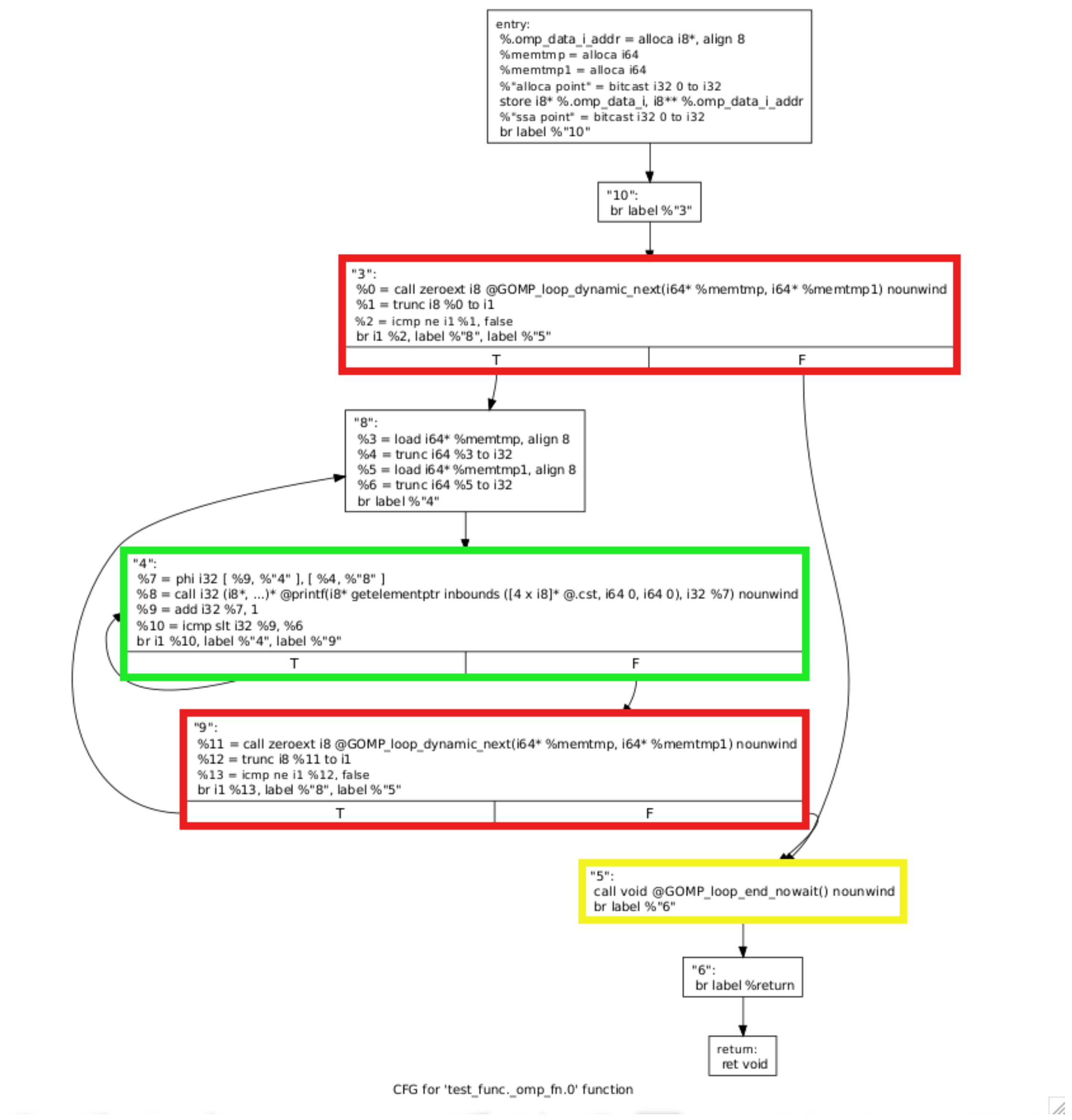
The access functions for the 2-dimensional array A are $A_1(i, j) = i + j$ and $A_2(k) = k + 1$. Substituting the UB and LB values of the iterators inside the expressions produces the UB and LB values for the access functions that are $UB_1 = 1$ $LB_1 = 0$ for dimension 1 and $UB_2 = 3$ $LB_2 = 1$ for dimension 2. For loop iterator *"i"*, zero has been used instead of the actual UB and LB values.

5.3.4 Slices and Ranges computation

Once the UB and LB have been computed for each array dimension we can compute the **slices** size and **range** intervals. This can be done easily by executing these two simple passes once for each array dimension:

1. If the dimension being considered is iterated through the most external loop iterator then the value computed is a **slice** of size $Slice = UB - LB + 1$.
2. If the dimension is not a slice then it is a **range**. The interval bounds are the same UB and LB of the array dimension.

```
int test_func() {
int i;
#pragma omp parallel for private(i) schedule(dynamic)
  for (i = 0; i < 10; ++i)
    printf("%d\n", i);
  return 0;
}
```



```
entry:
 %.omp_data_i_addr = alloca i8*, align 8
 %memtmp = alloca i64
 %memtmp1 = alloca i64
 %"alloca point" = bitcast i32 0 to i32
 store i8* %.omp_data_i, i8** %.omp_data_i_addr
 %"ssa point" = bitcast i32 0 to i32
 br label %"10"

"10":
 br label %"3"

"3":
 %0 = call zeroext i8 @GOMP_loop_dynamic_next(i64* %memtmp, i64* %memtmp1) nounwind
 %1 = trunc i8 %0 to i1
 %2 = icmp ne i1 %1, false
 br i1 %2, label %"8", label %"5"

"8":
 %3 = load i64* %memtmp, align 8
 %4 = trunc i64 %3 to i32
 %5 = load i64* %memtmp1, align 8
 %6 = trunc i64 %5 to i32
 br label %"4"

"4":
 %7 = phi i32 [ %9, %"4" ], [ %4, %"8" ]
 %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.cst, i64 0, i64 0), i32 %7) nounwind
 %9 = add i32 %7, 1
 %10 = icmp slt i32 %9, %6
 br i1 %10, label %"4", label %"9"

"9":
 %11 = call zeroext i8 @GOMP_loop_dynamic_next(i64* %memtmp, i64* %memtmp1) nounwind
 %12 = trunc i8 %11 to i1
 %13 = icmp ne i1 %12, false
 br i1 %13, label %"8", label %"5"

"5":
 call void @GOMP_loop_end_nowait() nounwind
 br label %"6"

"6":
 br label %return

return:
 ret void
```

Figure 5.2: Example of translated OpenMP loop with dynamic scheduling into LLVM-IR

```
int i,j,k, A[6][5];
#pragma omp parallel for private(i,j,k) schedule(dynamic)
for (i = 0; i < 6; i+=2)
  for (j = 0; j < 2; ++j)
    for (k = 0; k < 3; ++k)
      A[i+j][k+1] = 0;
```

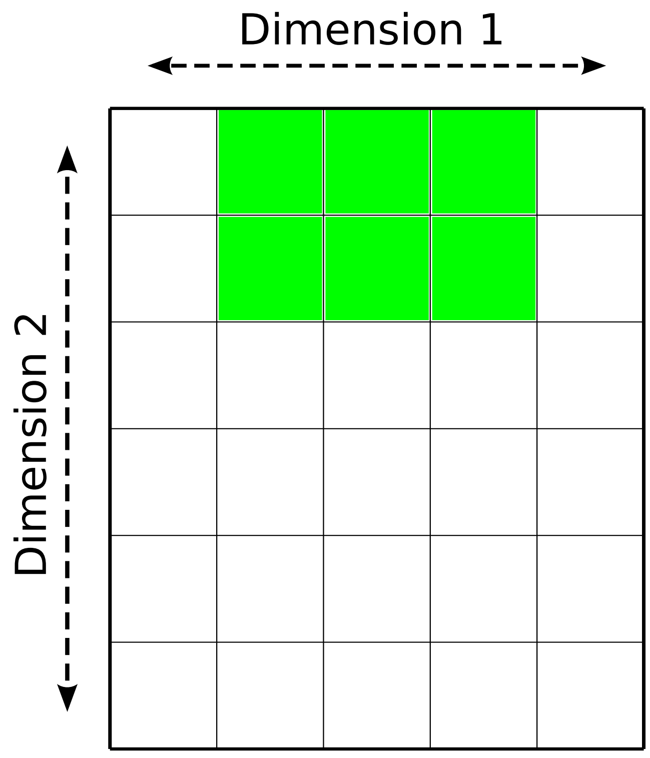


Figure 5.3: Example of access pattern on a 2-dimensional array

```
int i, n, *A;
n = returnsIntFunc();
for (i = 0; i < n; ++i) {
  A[i] = 0; //Statement 1
  if (i < 5) //This is a branch that creates a new basic block, hence a new statement
    A[i] = 1; //Statement 2
}
```

Figure 5.4: Example of simple parametric bound loop with two statements

Chapter 6

Polly Analysis Relaxation

Polly is the polyhedral analysis tool used to analyze the code and extract SCoP information from source code. Polly is aimed at analysis, transformation and code generation. Code generation means being able to translate a loop nest , represented in polyhedral form, back into proper LLVM-IR code. The need for code generation capabilities forces Polly into being very strict about which kinds of SCoPs it accepts for analysis. At its current level Polly is too strict for being used as an analysis tool. The main limits that have been found while using Polly for this project are the following:

- **Polly discards all SCoPs containing non-affine memory accesses.** This means that even if the SCoP being analyzed contains affine memory accesses that could be analyzed it is discarded anyway. The reason behind this is that non-affine accesses pose potential problems for memory dependence analysis. [13]

- **Polly discards all SCoPs containing non-affine branches.** Branches add to a SCoP statement domain additional constraints. All the constraints that can be added to polytope domains must be affine as a requirement. This is a problem that affects in particular code generation. Solutions for this has been target of studies and a solution exist using *Control Predicates* in order to embed the affine branch inside the statement itself [7]. This solution is, however, not suitable for analysis purposes, because it targets specifically code generation, which is not needed for this project.

- **Polly discards all SCoPs containing any type of cast expression.** Cast expressions introduce ambiguities to memory accesses. Memory aliasing becomes an issue when cast expressions are present. The compiler frontend used to compile C code into LLVM-IR generates many LLVM *bitcast* instructions inside OpenMP thread routines generated for the implementation of parallel blocks. One Bitcast instruction is added by the compiler for each memory access, invalidating almost all the SCoPs in an OpenMP worker function. The compiler also generates LLVM *trunc* instructions when dealing with indexing through loop iterators. Trunc is another kind of cast instruction that converts a type into a smaller one (fewer amount of bits).

To enable Polly accepting these cases it has to be modified accordingly and to do so it is needed to know about the process Polly uses to collect data.

6.1 Polly SCoP Extraction Process

Polly collects SCoP information in three phases like it is represented in Figure 6.1. Each phase passes information to the next in a pipeline manner. The steps are:

1. Identifying a SCoP inside the source code.
2. Collecting control-flow information about the identified SCoP
3. Constructing the polyhedral representation of the SCoP.

6.1.1 Polly SCoP Detection Process

The first step is to identify a Region comprising the SCoP. A **Region** is defined as *"... a connected sub-graph of a control flow graph that has exactly two connections to the remaining graph. It can be used to analyze or optimize parts of the control flow graph"* [14] . The nodes of this graph are basic blocks. The objective of this pass is to find regions that are compatible with the definition of SCoP and that also respect the additional limitations imposed by Polly. The *Scop Detection* pass of Polly is implemented as a single LLVM FunctionPass.

This algorithm (presented in pseudocode) is executed for each function in the code:

```
ScopList S;
findScops(Region R) {
  if (isValidScop(R)) {
    S.add(R);
    return;
  }

  foreach (SubRegion in Sub-Regions of R) {
      findScops(SubRegion);
}



bool isValidScop(Region R) {
  if (R is a top-level Region)
    return false;
  if (R is not a simple Region)
    return false;
  foreach (BasicBlock BB in R) {
    if (BB.Terminator is ReturnInstruction)
      return false;
    if (Branch condition for BB is non-affine)
      return false;
    //Invalid instructions are casts, function calls ...
    if (BB contains invalid instructions)
      return false;
    if (BB is part of a loop)
      if (Loop of BB has non-affine bounds)
        return false;
  }
  return true;
}
```

Polly starts by taking the top-level region of each function in the program and checks if the subregions contained in it are valid SCoPs regions. There are several rules that a SCoP has to obey to be accepted. Top-level regions (a region representing the whole function) are not accepted as SCoPs. Non-simple regions (regions connected to the control flow graph by more than two edges) also are discarded. Then each basic block of the region is checked. The branch exiting from the block can't be a return instruction and the condition of the branch, if it is a conditional branch, must be affine. The branch must also not contain certain instructions, like call and cast instructions. All access to memory (using LLVM load and store instructions) must be done through affine access functions. If the block is part of a loop the loop must have affine bounds. If just one of the blocks does not respect all the rules then the region is not suitable to be a SCoP region. It is then splitted in its subregions and the algorithm is repeated on each of the subregions. In this way Polly can find smaller suitable regions. The regions that are compatible with polyhedral analysis are then passed to the next step.

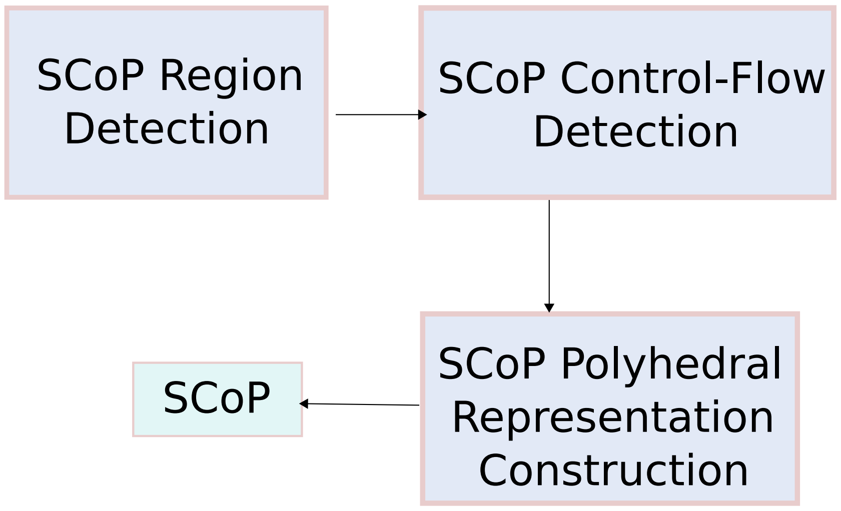

SCoP Region Detection
SCoP Control-Flow Detection
SCoP Polyhedral Representation Construction
SCoP

Figure 6.1: Polly SCoP detection process

6.1.2 Polly SCoP Control-Flow Analysis

This pass is aimed at gathering information about the SCoP to construct its polyhedral representation in the latest pass. This process is implemented as an LLVM FunctionPass that relies on the detection pass for SCoP region definitions. The detection pass provides as output a list with all the regions suitable to be SCoPs. The main data that are going to be gathered in this pass are:

1. Memory access functions

2. Basic-Blocks branch conditions

3. Bounds of loops contained in the SCoPs

Memory access functions. Information about Memory access functions are stored as SCEV (Scalar Evolution) expressions. These are simply gathered through the SCEV engine that LLVM provides. Originally only affine access functions were expected to be found in this phase. A SCEV expression represents an analyzed expression in a program. It can represent an unknown value (like a program variable) or a constant (a value known at compile time) or an operation over these, like arithmetic or cast operators. SCEV expressions representing operations have other SCEV expressions as operands. Depending on the type of the operation the expression may have one or multiple operands. The complete inheritance graph for the classes that represent SCEV expressions in LLVM can be found in Figure 6.2.[16] SCEV expressions can be visited and analyzed using a *Visitor Pattern* [17], expoiting the class hierarchy of the SCEV classes. LLVM provides also a template class in order to simplify the development of SCEV visitors. It is called *SCEVVisitor*, and to construct a SCEV expander through this template is enough to just inherit from it and define visit methods for each type of SCEV expression that LLVM supports. Each of these visit method is called *visit*()* where * is a placeholder for the type of the SCEV that the method handles (for example *visitConstant()* is the method that handles constant SCEV expressions). The types of SCEV classes supported by LLVM can be seen in Figure 6.2. Calling then the generic *visit()* method with a SCEV object as parameter will automatically call the right handling method depending on the actual type of the SCEV class being analyzed.

Branch conditions. Branch conditions for a basic block are gathered by constructing the dominator tree of the basic block being considered (more info about Dominator Trees in LLVM can be found in Section 7.4.2, paragraph *The Dominator tree*). The dominator tree is then walked from the basic-block back toward the entry point of the region (the entry block of the region is unique being the region simple) using the immediate dominators of the tree nodes. Every condition found on the path (in the form of a SCEV expression connected to a compare instruction used by the branch terminators of a basic block) is analyzed and the condition and predication type are extracted and stored. In the example of Figure 6.3 we have two basic-blocks, one that executes the assignment $b = 5$ and one that executes $c = 6$. The former basic block has an execution condition of $a > 5$, while the latter has two execution conditions that have to be true at the same time: $a > 5$ and $a < 10$. This pass stores for each basic-block all the conditions that must be true for it to execute. Each condition expression has an associated predicate (the equality sign of the expression), a right hand side expression and a left hand side expression. In the case of $a > 5$ ">" is the predicate, *"a"* is the left hand side expression while *"5"* is the right hand side. These are stored as SCEV expressions.

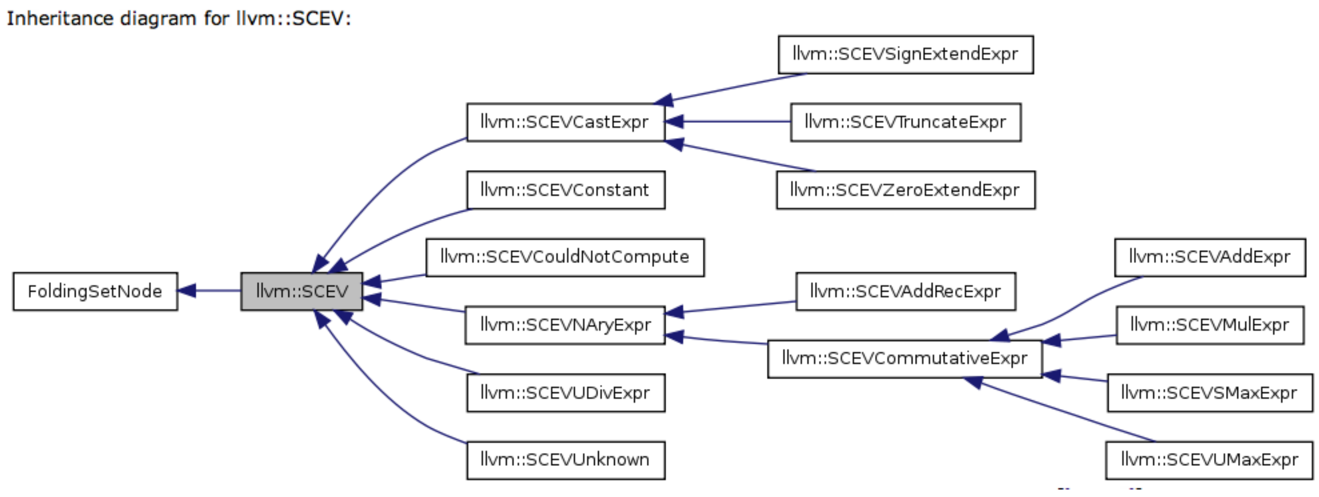


Figure 6.2: Inheritance graph for LLVM SCEV class

Loop bounds. Loop bounds are gathered through the Scalar Evolution engine of LLVM using a tool the engine itself provides that computes the backedge taken count of a loop (amount of times a loop executes). The loops accepted by Polly are only normalized loops (going from zero to a certain value, the *BackedgeTakenValue* in this case). *BackedgeTakenValue* is a SCEV expression, not necessarly a constant value.

```
int a, b, c;
if (a > 5) {
  b = 5;  // This is executed if a > 5
  if (a < 10) {
    c = 6;  // This is executed if a > 5 and a < 10
  }
}
```

Figure 6.3: Example of conditional code

6.1.3 Polly SCoP Polyhedral representation construction

After all the data has been gathered Polly proceeds into constructing the polyhedral representation of the SCoP transforming the data into ISL data structures. For each basic block of LLVM-IR code in a SCoP, Polly constructs a SCoP statement. Each

SCoP statement has a domain , a scattering vector and a certain number of memory accesses. Polly begins by constructing the *space* in which the SCoP statement lives. The space specifies the amount of induction variables and parameters on which the SCoP statement depends on. All the subsequent data structures containing polyhedral information are constructed on the space created here. From the branch conditions gathered in the previous pass Polly constructs the domain of the SCoP statement. The domain is the collection of values of the SCoP induction variables or parameters for which the basic block of the SCoP statement is executed. It is represented as an integer-set of points , living in the SCoP space, using the *isl_set* data structure. Then the scattering is constructed. The scattering represents the schedule of the SCoP statement in respect to the others. It is devised from the CFG of the program and is represented using a relation. The relation is between the induction variables of the SCoP loops and a scattering vector representing the schedule of the SCoP statement (more info about scattering vectors in Section 3.1.1). This relation is expressed in code using an *isl_map* data-structure. At last the memory accesses are constructed. For each memory access contained in the statement Polly analyzes the SCEV expression gathered in the previous pass and transforms it in an affine expression using a *SCEVVisitor* class. It constructs an *isl_aff* object representing this access and then converts it into an *isl_map*. All this data is stored in a class called *Scop* that can then be used by transformation or analysis passes to query about polyhedral information.

6.2 Ignoring Non-affine memory accesses

Non-affine memory accesses are not supported directly by Polly and a SCoP containing any number of these accesses is immediately discarded. To overcome this limitation Polly has been modified so that instead of discarding SCoPs containing non-affine memory accesses it ignores and flags them as "non-affine" , continuing the analysis. Ignoring non-affine memory accesses inside SCoPs may be an unwanted behavior. To avoid this becoming the default behaviour of Polly it has been decided (after discussing with Polly main developer) that the best approach would be adding a flag that can be checked by Polly when the passes are run. If the flag is present Polly accepts SCoPs containing non-affine memory accesses marking them as *"potentially accessing the whole memory space"*. if the flag is not present Polly default behaviour remains the same. The result of this work has being tested and transformed into a patch that has been integrated into LLVM in December 2011 [15].

6.2.1 Changes to the Detection pass

During the detection pass, when a LLVM load or store instruction is found, the access function (in the form of a SCEV expression) is analyzed and is determined if it is affine or not. A SCEVVisitor class is used for this. Normally if it is affine then it is accepted and if it isn't the whole SCoP containing it gets discarded. After the change the detection pass does not discard the SCoP anymore and both it and the memory access get accepted to the next step.

6.2.2 Changes to the Control-flow analysis pass

During the control-flow analysis pass memory access information is gathered and provided to the next pass. Polly didn't support non-affine memory accesses and didn't expect them in this phase so all of the memory accesses gathered were automatically flagged as affine. After the change the access function is checked for affinity again, and if it isn't affine then this condition is notified to the next pass.

6.2.3 Changes to the Polyhedral construction pass

This pass receives the description of memory accesses from the previous pass and constructs their polyhedral representation. Non-affine accesses cannot have a polyhedral representation. If the memory access being constructed isn't affine then the relation that describes the memory access is set to the universe relation. For the affine memory accesses their polyhedral representation is constructed normally.

6.3 Ignoring Non-Affine branches

Non-Affine branches are not supported by Polly. Polly needs to construct the domain of a SCoP statement from the affine conditions that need to be satisfied to execute that statement. If one of these condition is non-affine Polly can't do that and discards the whole SCoP, because it can't completely represent them in polyhedral from. The solution that has been implemented is to transform any non-affine condition into a true condition. This has the same effect as removing the condition itself. Because our analysis does not require to be an exact analysis, but aims to generate only a tip for the compiler this is a valid solution. The same approach used for non-affine memory accesses has been used by adding a boolean flag to Polly that, when enabled, enables the new functionality.

6.3.1 Changes to the Detection pass

During the detection pass, Polly checks for the affinity of basic blocks terminator conditions in the SCoP and it discards all the SCoPs containing block terminators that depend on non-affine conditions. To make this check Polly calls a function called *isAffineExpr()* that takes the SCEV expression representing the basic-block terminator condition and extracts its composing parts checking that all the conditions for affinity are met. Polly *isAffineExpr()* function simply calls the *visit()* method of an instance to a *SCEVValidator* class. *SCEVValidator* is a class that inherits from the LLVM *SCEVVisitor* template and is used to analyze and validate the SCEV expression. Depending on the type of SCEV expression that is passed to the visit method, one of the many handling method is called. Should the type of the SCEV expression be a value (like an unknown value or a constant) then its affinity is evaluated and the method returns the result as a boolean. Should the type be instead an operational SCEV expression (like an arithmetic operation composed of many operands or a cast expression) then the handling method calls recursively the visit method once for each of the operands of the expression, checking that each of them is affine. An expression is affine only if each of the operands that compose it are affine themselves. The class diagram for the *SCEVValidator* class can be found in Figure 6.4.

Polly requires that some extra conditions are met other then the standard SCoP affinity constraints explained in Chapter 3. Cast expressions are not allowed and neither are unsigned operations, like unsigned multiplication or division (at the moment Polly does not support unsigned operations). It is still required , then, to discard SCoPs that have conditions that contain casts or unsigned operations. To do so a new *SCEVVisitor* class has been created that just checks that no unsigned operation or cast expressions are contained in the SCEV expression, without checking for the actual affinity of the expression. The new class has been called *BrSCEVValidator*, while the new check function (that uses this new visitor class) is called *isAffineExprBr()*. The new check function is used instead of the original one if the flag is enabled.

Here is presented the pseudo-code for the SCEV expressions handling functions of the *BrSCEVValidator* class:

```
//The visit() method calls one of the appropriate handling
//method below depending on the real type of the SCEV expression

AffinityResult visitConstant(SCEVConstant *Constant) {
//Constants are always affine
  return CONSTANT;
}

AffinityResult visitTruncateExpr(SCEVTruncateExpr *Expr) {
  AffinityResult result = visit(Expr->getOperand());
  return result;
}

bool visitZeroExtendExpr(SCEVZeroExtendExpr *Expr) {
  AffinityResult result = visit(Expr->getOperand());
  //Accept only if the operand of the cast is constant
  if (result is a CONSTANT)
    return CONSTANT;

  return INVALID;
}

AffinityResult visitSignExtendExpr(SCEVSignExtendExpr *Expr) {
  //Sign extend is a NO-OP, invalid only if the operand is unsigned
  AffinityResult Op = visit(Expr->getOperand());

  return Op;
}
```

```
AffinityResult visitAddExpr(SCEVAddExpr *Expr) {
  AffinityResult Return = CONSTANT;
  foreach (SCEV *ExprOperand in Expr->getOperands()) {
    AffinityResult result = visit(ExprOperand);
    if (result is INVALID)
      return INVALID;
    //Update the type of the result
    if (Return is CONSTANT and (result is a PARAMETER or
                        result is an INDUCTION_VARIABLE))
      Return = result;
    if (Return is a PARAMETER and result is an INDUCTION_VARIABLE)
      Return = result
  }
  return Return;
}

AffinityResult visitMulExpr(SCEVMulExpr *Expr) {
  AffinityResult Return = CONSTANT;
  foreach (SCEV *ExprOperand in Expr->getOperands()) {
    AffinityResult result = visit(ExprOperand);
    if (result is CONSTANT)
      continue;
    //Multiplication between more than one variable values are not supported
    if (Return is not CONSTANT)
      return INVALID;
    //Update the type of the return
    if (Return is CONSTANT and (result is a PARAMETER or
                        result is an INDUCTION_VARIABLE))
      Return = result;
    if (Return is a PARAMETER and result is an INDUCTION_VARIABLE)
      Return = result
  }
  return Return;
}
```

```
AffinityResult visitUDivExpr(SCEVUDivExpr *Expr) {
  AffinityResult LHS = visit(Expr->getLHS());
  AffinityResult RHS = visit(Expr->getRHS());

  //Only divisions between constants are supported (treated as parameters)
  if (LHS is CONSTANT and RHS is CONSTANT)
    return PARAMETER;

  return INVALID;
}

AffinityResult visitAddRecExpr(SCEVAddRecExpr *Expr) {
  if (Expr is not Affine) {
    return INVALID;
}

  AffinityResult Start = visit(Expr->getStart());
  AffinityResult Recurrence = visit(Expr->getStepRecurrence());

  //Only recurrent expressions with affine start
  //and constant recurrence are supported
  if (Start is INVALID or Recurrence is not CONSTANT)
    return INVALID;

  if (SCOPRegion contains Expr->getLoop())
    return INDUCTION_VARIABLE;

  if (Start is CONSTANT)
    return PARAMETER;

  return INVALID;
}
```

```
AffinityResult visitSMaxExpr(const SCEVSMaxExpr *Expr) {
  ValidatorResult Return = CONSTANT;
  foreach (SCEV *ExprOperand in Expr->getOperands()) {
    ValidatorResult Result = visit(ExprOperand);
    if (Result is INVALID)
      return INVALID;
    //Update the type of the return
    if (Return is CONSTANT and (result is a PARAMETER or
                          result is an INDUCTION_VARIABLE))
      Return = result;
    if (Return is a PARAMETER and result is an INDUCTION_VARIABLE)
      Return = result
  }
  return Return;
}

AffinityResult visitUMaxExpr(SCEVUMaxExpr *Expr) {
  ValidatorResult Return = PARAMETER;
  foreach (SCEV *ExprOperand in Expr->getOperands()) {
    ValidatorResult Op = visit(ExprOperand);
    if (Op is INVALID)
      return INVALID;
    //Update the type of the return
    if (Return is CONSTANT and (result is a PARAMETER or
                          result is an INDUCTION_VARIABLE))
      Return = result;
    if (Return is a PARAMETER and result is an INDUCTION_VARIABLE)
      Return = result
  }
  return Return;
}

AffinityResult visitUnknown(const SCEVUnknown *Expr) {
  //Unknowns treated as parameters
  return PARAMETER;
}
```

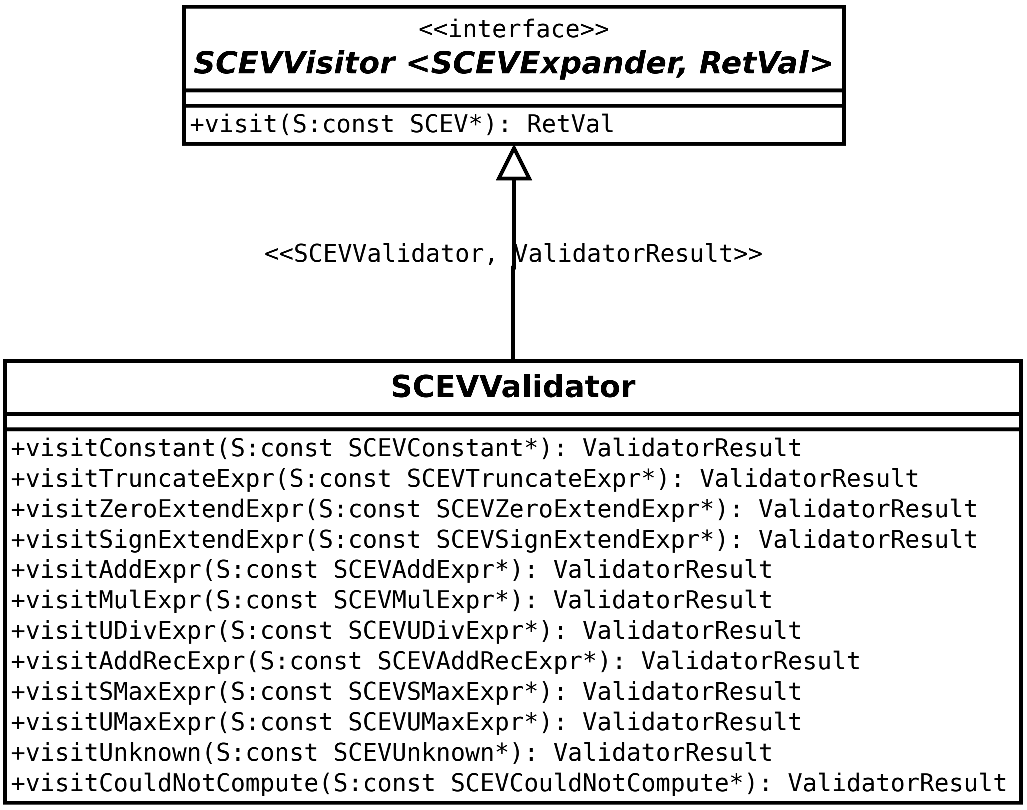

<<interface>>
SCEVVisitor <SCEVExpander, RetVal>
+visit(S:const SCEV*): RetVal
<<SCEVValidator, ValidatorResult>>
SCEVValidator
+visitConstant(S:const SCEVConstant*): ValidatorResult
+visitTruncateExpr(S:const SCEVTruncateExpr*): ValidatorResult
+visitZeroExtendExpr(S:const SCEVZeroExtendExpr*): ValidatorResult
+visitSignExtendExpr(S:const SCEVSignExtendExpr*): ValidatorResult
+visitAddExpr(S:const SCEVAddExpr*): ValidatorResult
+visitMulExpr(S:const SCEVMulExpr*): ValidatorResult
+visitUDivExpr(S:const SCEVUDivExpr*): ValidatorResult
+visitAddRecExpr(S:const SCEVAddRecExpr*): ValidatorResult
+visitSMaxExpr(S:const SCEVSMaxExpr*): ValidatorResult
+visitUMaxExpr(S:const SCEVUMaxExpr*): ValidatorResult
+visitUnknown(S:const SCEVUnknown*): ValidatorResult
+visitCouldNotCompute(S:const SCEVCouldNotCompute*): ValidatorResult

Figure 6.4: Inheritance graph for LLVM SCEV class

6.3.2 Changes to the Control-flow analysis pass

No changes are needed to the control-flow analysis, because this pass just collects the SCEV expressions of the execution conditions for each basic-block. No analysis of the SCEV expressions occur in this pass, therefore no special treatment for affine or non-affine expressions is needed.

6.3.3 Changes to the Polyhedral construction pass

This pass constructs the polyhedral description of the execution conditions of the SCoP statements and merge them to generate the domain of the SCoP statements. Non-affine conditions cannot be transformed into polyhedral form and added to the domain, so these need special handling. When a new SCoP statement is to be constructed each execution condition gathered at the previous pass is transformed to polyhedral form. Non-affine conditions must be identified and transformed into an *always true* condition (like 1 == 1 for example). A condition is composed of two parts: a *left hand side* part (LHS) and a *right end side* part (RHS). The two parts are connected together by the predicate of the condition. If the flag is enabled then the affinity of the two parts is checked and , if one of them is not affine an universe set (equivalent to an always true condition) is constructed and returned. If the condition is affine, instead, the polyhedral representation of the two parts is constructed and they are composed together constructing the truth set of the condition. The pseudo-code for this part of the pass is:

```
PolyhedralAffineExp L, R;
if (NonAffineBranchFlag is true) {
  if (LeftHandSidePart is not affine or RightHandSidePart is not affine) {
    return UniverseSet;
  }
}

L = Polyhedral representation of the LeftHandSidePart;
R = Polyhedral representation of the RightHandSidePart;

return composeCondition(L,R, ConditionPredicate);
```

6.4 Dealing with casts in OpenMP loops

As stated at the beginning of this chapter Polly does not support any kind of cast expressions inside the body of a SCoP. LLVM provides many different kinds of cast instructions that can be used inside LLVM-IR code to convert the type of some memory pointers to different types. The types of cast instructions found in LLVM are:

- **Trunc Instructions.** These are used to truncate the integer or floating-point types to a smaller type. For example it is possible to truncate a 64-bit integer to a 32-bit integer. Truncate may generate information loss, because truncation to a smaller type may alter the values of the data accessed. The Trunc instruction has both an integer and a floating-point version.
- **Bit-Cast Instructions.** These are no-op casts that simply reinterpret the type of a memory pointer from a type to another.
- **Zero Extension Instructions.** These are instructions that convert integer or floating-point types to bigger types by filling the additional bits with zeroes. There are both an integer and a floating-point version of this instruction.
- **Sign Extension Instructions.** These are instructions that convert integer or floating-point types to bigger types by filling the additional bits with the sign bit of the original value. There are both an integer and a floating-point version of this instruction.
- **Floating-point to integer conversion.** These instructions convert floating-point types to integer types. It is possible to convert to both signed and unsigned integer types.
- **Integer to Floating-point conversion.** These instructions convert signed or unsigned integer types to floating-point types.
- **Floating-point Extension Instruction.** This instruction converts a floating-point type to a bigger one.
- **Integer to pointer and pointer to integer conversion.** These instruction convert an integer to a pointer type and vice versa.

A class diagram for LLVM cast instructions can be found in Figure 6.5.

The OpenMP compiler generates some cast instructions inside the code , in particular **Trunc** and **Bit Cast** instructions. Trunc instructions are generated when dealing with loop iterators, while Bit Casts are generated for each memory access inside the generated parallel OpenMP function (more info about the OpenMP loop translation process can be found in section 5.1). This produces SCEV expressions for memory access functions and SCoP statements execution conditions containing casts not supported by Polly. This prevents Polly from analyzing successfully most of the SCoPs containing an OpenMP loop.

What is needed is finding a way to deal with these newly introduced cast instructions, in order to make Polly accept the SCoP anyway, even if the compiler introduces these casts.

A first observation that can be made about Trunc casts is that, because we are not interested in code generation, these casts can be pretty much ignored and treated as no-ops. In the case a trunc instruction actually causes the overflow of an index ,for example , it may produce a wrong analysis of the loop structure, but this is a very rare event. The only consequence of this happening is a non optimal analysis.

Bitcasts are more complicated, because in order to allow them into Polly it would be required to make deep changes to the core of Polly itself. It has been decided then to take another approach to this problem by removing the bitcasts introduced by the compiler through a transformation pass that takes as input the generated compiled LLVM-IR code and outputs normalized code with the bitcasts removed.

6.4.1 Dealing with Trunc instructions

As stated above, to deal with Trunc instructions it is required to make Polly accept code containing Trunc instructions and treat them as no-ops. Trunc instructions are represented in SCEV form as a Trunc SCEV expression (using the class *SCEVTruncateExpr*). The original *SCEVValidator* used by Polly to validate SCEV expressions discards any SCoP containing Trunc constructs (and any other kind of cast expression). We already created a new *SCEVValidator* as part of the solution for non-affine branches (check Section 6.3 for more information), to make Polly accept Trunc expressions it is enough to make our new *BrSCEVValidator* class ignore them, the result of the analysis of the Trunc instruction becomes the result of the analysis on the operand. The pseudo-code presented in Section 6.3 already has this modification integrated.

6.4.2 Dealing with Bit-Casts

Bitcasts can't be ignored. The strategy used to solve the problem of compiler generated bit-casts consists in removing the bit-casts from the code itself using

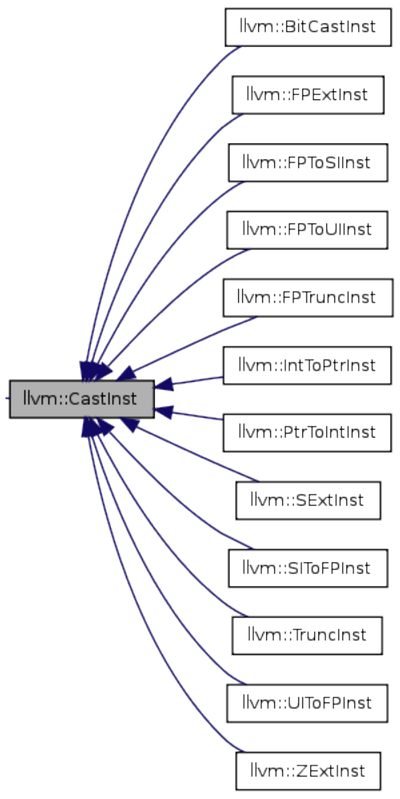


Figure 6.5: LLVM cast instructions

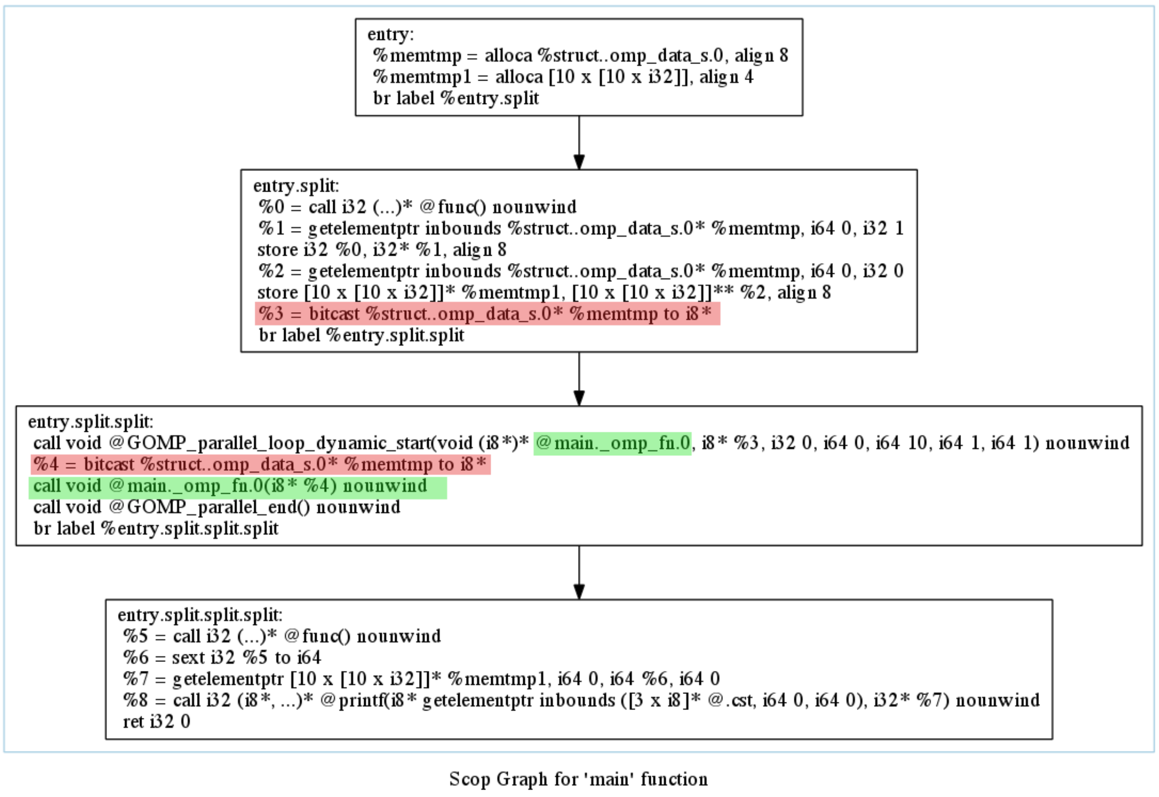
a peculiarity on how the compiler translates OpenMP C code into LLVM-IR.

Details on OpenMP loop translation When an OpenMP parallel directive is translated to LLVM-IR the compiler creates a new function that contains the code of the parallel section to execute. This function is used as the thread routine that each new spawned thread executes. To pass arguments to the parallel code (like pointers to shared data structures) it is used a structure data type. A new structured type is created for each parallel function that the compiler generates. For example if the parallel block accepts as input a 32-bit integer value and the pointer to an array of 10 32-bit integers then the type generated will look like this (in LLVM-IR code):

```
//The syntax to define named aggregate types in LLVM-IR is
//%typename = type { list of types separated by comma }
%struct..omp_data_s.0 = type { [10 x i32]*, i32 }
```

A pointer to an instance of this type is passed as argument to the generated function. The function then uses this to access the shared data. An example of the code generated from the compiler for a parallel block is shown in Figure 6.6. The

OpenMP parallel function is highlighted in green, while the shared data structures that are passed to the function as parameters are highlighted in red. The generated function is called both by the main thread of the program and from the new threads that are spawned using the *GOMP_parallel_loop_dynamic_start()* function.



```
entry:
 %memtmp = alloca %struct..omp_data_s.0, align 8
 %memtmp1 = alloca [10 x [10 x i32]], align 4
 br label %entry.split
```

```
entry.split:
 %0 = call i32 (...)* @func() nounwind
 %1 = getelementptr inbounds %struct..omp_data_s.0* %memtmp, i64 0, i32 1
 store i32 %0, i32* %1, align 8
 %2 = getelementptr inbounds %struct..omp_data_s.0* %memtmp, i64 0, i32 0
 store [10 x [10 x i32]]* %memtmp1, [10 x [10 x i32]]** %2, align 8
 %3 = bitcast %struct..omp_data_s.0* %memtmp to i8*
 br label %entry.split.split
```

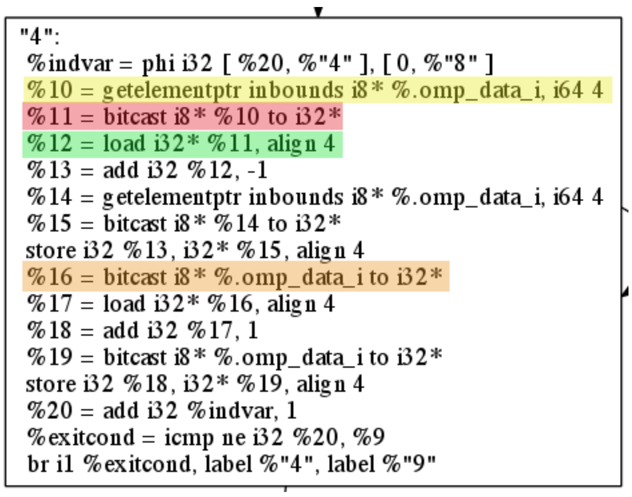
```
entry.split.split:
 call void @GOMP_parallel_loop_dynamic_start(void (i8*)* @main._omp_fn.0, i8* %3, i32 0, i64 0, i64 10, i64 1, i64 1) nounwind
 %4 = bitcast %struct..omp_data_s.0* %memtmp to i8*
 call void @main._omp_fn.0(i8* %4) nounwind
 call void @GOMP_parallel_end() nounwind
 br label %entry.split.split.split
```

```
entry.split.split.split:
 %5 = call i32 (...)* @func() nounwind
 %6 = sext i32 %5 to i64
 %7 = getelementptr [10 x [10 x i32]]* %memtmp1, i64 0, i64 %6, i64 0
 %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.cst, i64 0, i64 0), i32* %7) nounwind
 ret i32 0
```

Figure 6.6: Example of code generated from a parallel block

What the compiler does is creating a function that accepts as parameter a *byte pointer* (i8* in LLVM-IR). Then the data structure , when passed to the function, is bitcasted from its actual type to *i8**. This kind of pointer acts similarly to a void pointer in C, that is used as a generic pointer type. In the function, when data is accessed through the data structure, the *i8 pointer* is casted back to the actual type contained in the data structure just before being used in a load or store instruction. An example of this is shown in Figure 6.7. Here the memory access to shared data is shown in green , the bitcast that convert back the data from an *i8 pointer* to the actual type is shown in red.

Bitcast removal process The compiler generates bitcasts for each load and store instruction because it needs to convert the *i8 pointer* to the actual data type of the memory accessed. If instead of accepting an *i8 pointer* the generated function



```
"4":
 %indvar = phi i32 [ %20, %"4" ], [ 0, %"8" ]
 %10 = getelementptr inbounds i8* %.omp_data_i, i64 4
 %11 = bitcast i8* %10 to i32*
 %12 = load i32* %11, align 4
 %13 = add i32 %12, -1
 %14 = getelementptr inbounds i8* %.omp_data_i, i64 4
 %15 = bitcast i8* %14 to i32*
 store i32 %13, i32* %15, align 4
 %16 = bitcast i8* %.omp_data_i to i32*
 %17 = load i32* %16, align 4
 %18 = add i32 %17, 1
 %19 = bitcast i8* %.omp_data_i to i32*
 store i32 %18, i32* %19, align 4
 %20 = add i32 %indvar, 1
 %exitcond = icmp ne i32 %20, %9
 br i1 %exitcond, label %"4", label %"9"
```

Figure 6.7: Example of memory access in the generated OpenMP parallel function

accepted a pointer of the correct type then the bitcasts wouldn't be necessary and could be removed. The solution that has been used is creating a new function with a signature that accepts as parameter a pointer to the actual structure data type (instead of an *i8 pointer*) and cloning the code of the compiler generated function into this new function. The complete step list for this process is:

1. Add a bitcast at the entry block of the OpenMP generated function converting the function argument from a generic *i8 pointer* to the actual structure data type.
2. Substitute all the usages of the function argument with new instructions referring to the newly generated bitcast instruction.
3. Create a new function signature for the OpenMP function accepting as parameter a pointer to the actual data type.
4. Clone the original function code into this newly generated function.
5. Substitute all the usages of the bitcast instruction introduced at point 1 with the function argument and remove the bitcast.
6. Make all the references to the old OpenMP function point to the new one instead.
7. Remove the old OpenMP function.

In point number 2 the usages of the function argument, at this stage, can only be of two kinds. The first kind is Bitcast instructions. Bitcast instructions are used to obtain a pointer of the real type of the data that is going to be accessed. The second kind is *GetElementPtr* instructions [18]. *GetElementPtr* instructions are used in LLVM to do pointer arithmetics. Using *GetElementPtr* instructions it is possible to address arbitrary elements inside a data type. This instruction does not make any memory access by itself, but it is just used to compute addresses. In Figure 6.7 the usage of a *GetElementPtr* instruction is in yellow. In this Figure the entire process of accessing shared data in an OpenMP parallel function has being highlighted in different colors. First the address of the variable being accessed is computed by the *GetElementPtr* instruction (yellow). In this case a variable distant 4 bytes from the beginning of the shared data structure is going to be accessed. After the computation of the target address, the type of the pointer returned from the *GetElementPtr* instruction is converted from *i8** to the actual data type being accessed through a bitcast (red). Then the memory access is performed through the pointer returned from the bitcast instruction (green). If the data that has to be accessed has offset zero from the beginning of the data structure the *GetElementPtr* instruction is skipped and the Bitcast is performed directly on the function argument (orange). There are, then, two different cases to handle:

1. The data that is going to be accessed has a non-zero offset from the beginning of the shared data structure. In this case a *GetElementPtr* instruction uses the function argument as operand to compute the address of the data and the successive Bitcast instruction uses the *GetElementPtr* as its operand

2. The data that is going to be accessed has a zero offset from the beginning of the shared data structure. In this case the *GetElementPtr* instruction is skipped and the Bitcast instruction uses directly the function argument as its operand.

In case number 1 it is needed to substitute the existing *GetElementPtr* instruction with a new one that uses as operand the newly added Bitcast instruction in step number 1. The indexing of the new *GetElementPtr* instruction is not computed in bytes as in the old one, but as an element offset inside the shared data structure. The Bitcast instruction can then be removed and all the references to the Bitcast instruction substituted with references to the new *GetElementPtr* instruction.

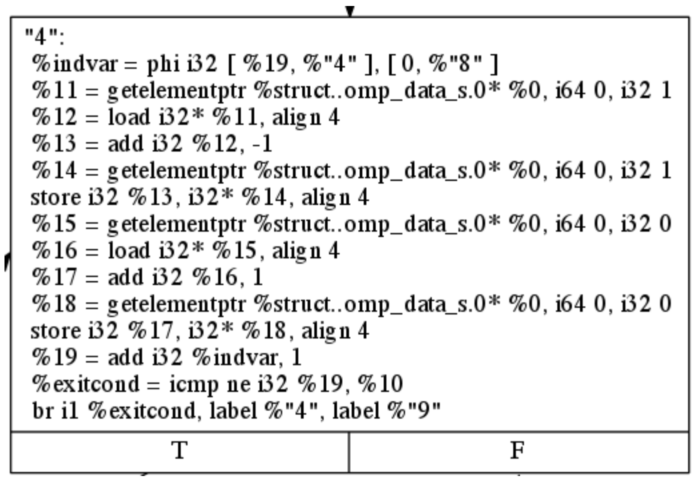
Case number 2 is easier. It is possible to directly remove the Bitcast instruction and substitute it with a *GetElementPtr* instruction pointing to the first element the shared data structure.

The whole procedure is implemented as a LLVM Transformation Module pass called **OpenMPCastRemoval**. A Module Pass is a kind of LLVM pass that works on the entire compilation unit as a whole. This is needed, because we need to work on more than one function at once. In particular we need to add new functions to the compilation unit and delete other ones, as well as modifying the code of functions referring to the functions we have removed. This pass also uses other two LLVM passes developed for this project that are the *OpenMPData* and *OpenMPRegionTree* passes. These are two analysis passes used to gather information for other passes. In particular the *OpenMPRegionTree* pass constructs a *Region Tree* that represents the connection between OpenMP compiler generated functions and the functions that call them (the ones containing the OpenMP parallel block). These passes are explained in greater detail in Chapter 7.

The pseudo-code for this transformation pass is presented in the next page.

```
node_vector = Empty vector of OpenMP Region Tree nodes;
foreach (RegionNode in OpenMP Region Tree)
  if (RegionNode.Function is OpenMP Generated Function)
    Add RegionNode to node_vector;
foreach (RegionNode in node_vector) {
  BitcastInst = New Bitcast At EntryPoint of RegionNode.Function
                from i8 to Actual Data Type;
  foreach (ArgUser in List Of Users Of RegionNode.Function Argument) {
    //Case 1
    if (ArgUser is GetElementPtr) {
      NewGetElementPtr = New GEP instruction That References BitcastInst;
      foreach (Bitcast in List Of Bitcasts using ArgUser) {
        Replace all uses of Bitcast with NewGetElementPtr;
        Erase Bitcast;
      }
      Erase ArgUser;
    } else {  //Case 2
      NewGetElementPtr = New GEP instruction that references first element
                         of structure pointed by BitcastInst;
      Replace all uses of ArgUser with NewGetElementPtr;
      Erase ArgUser;
    }
  }
  NewFunction = Create new function signature with
                a signle Actual Data Type pointer argument instead of i8*;
  Clone RegionNode.Function code into NewFunction;
  BitcastInst = Bitcast at the beginning of NewFunction;
  Replace all uses of BitcastInst with the argument of NewFunction;
  Erase BitcastInst;
  ParentNode = Parent Node of RegionNode;
  foreach (User of RegionNode.Function) {
    Replace all uses of RegionNode.Function in User with NewFunction;
    Pass the correct argument to User instead of a bitcasted argument to i8*;
  }
  Erase RegionNode.Function;
}
```

In Figure 6.8 it is possible to see how the code in Figure 6.7 changed after the transformation pass was enabled. It is easy to notice how all the bitcasts contained in the block have disappeared.



```
"4":
 %indvar = phi i32 [ %19, %"4" ], [ 0, %"8" ]
 %11 = getelementptr %struct..omp_data_s.0* %0, i64 0, i32 1
 %12 = load i32* %11, align 4
 %13 = add i32 %12, -1
 %14 = getelementptr %struct..omp_data_s.0* %0, i64 0, i32 1
 store i32 %13, i32* %14, align 4
 %15 = getelementptr %struct..omp_data_s.0* %0, i64 0, i32 0
 %16 = load i32* %15, align 4
 %17 = add i32 %16, 1
 %18 = getelementptr %struct..omp_data_s.0* %0, i64 0, i32 0
 store i32 %17, i32* %18, align 4
 %19 = add i32 %indvar, 1
 %exitcond = icmp ne i32 %19, %10
 br i1 %exitcond, label %"4", label %"9"
```

Figure 6.8: Example of memory access in the generated OpenMP parallel function with OpenMPCastRemoval

Chapter 7

Pattern Computation

The process of pattern computation is a multi step process that produces as output the patterns of all the affine memory accesses in OpenMP loops. The process is implemented as a set of LLVM analysis and transformation passes that work together like a pipeline (the output of one pass is the input for the next one). The types of these passes can be divided in three pass categories:

- Data pass
- Transformation pass
- Analysis pass

A **Data pass** has the purpose of maintaining data structures for the other passes. Data passes do not work on the code itself, but are simply containers of useful information for the other passes. **Transformation passes** modify the code, restructuring it, with the objective of making the successive analysis passes possible or easier to do. **Analysis passes** analyze the code and gather information to be used for later passes. These are the core of the project, because our objective is an analysis of the code. The final pass of the pipeline (*OMPNUMAIdentify*) is an analysis pass that produces the pattern information's as its output. The project is composed of a total of 6 passes:

1. **OpenMPData:** this is a *Data pass* whose purpose is to store LLVM data structures commonly used from the other passes. In particular this pass stores pointers to LLVM Function objects representing the OpenMP runtime functions (for example *GOMP_dynamic_loop_start()*) that can be found throughout the code that will be analyzed.

2. **OpenMPNorm:** this is a *Transformation pass* whose purpose is to apply common transformations to normalize the LLVM-IR code of the program so that subsequent passes can make certain assumptions about the structure of the code they will encounter.

3. **OpenMPCastRemoval:** this pass is a *Transformation pass* whose purpose is to remove bitcasts introduced by C compilers inside the LLVM-IR code. This pass is explained in detail in Section 6.4.2

4. **OpenMPDetection:** this is an *Analysis pass* whose purpose is to determine the position of OpenMP parallel and loop blocks inside the code.

5. **OpenMPRegionTree:** this is an *Analysis pass* whose purpose is to construct the *Region Tree* of the various parallel and loop blocks. The Region Tree defines the parent-child relationship between these blocks. The pattern construction pass uses this tree to find the regions to analyze.

6. **OMPNUMAIdentify:** this is an *Analysis pass* whose purpose is to build the accesses pattern data. This is the main pass of the set. All the other passes exist only to allow this pass computing the pattern information.

The relationship of usage between the various passes is represented in Figure 7.1. The *OpenMPCastRemoval* pass is not always used from the *OMPNUMAIdentify* pass, because not all compilers introduce bitcasts. For example the Fortran compiler does not and in that case *OpenMPCastRemoval* is not run.

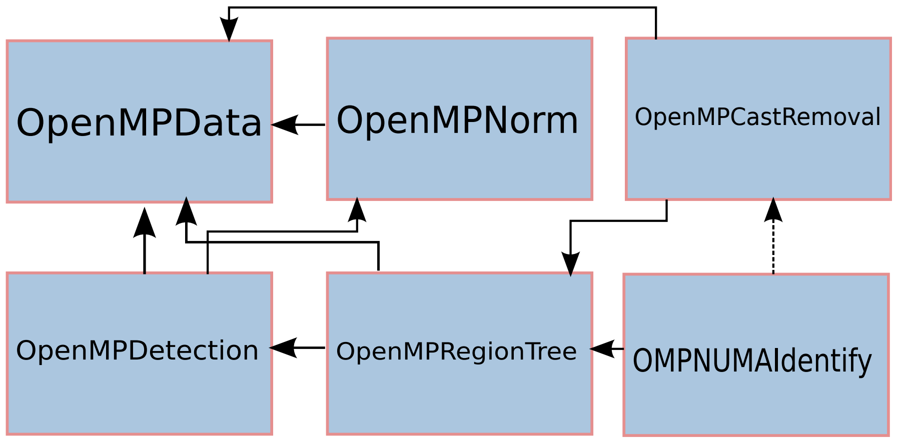


Figure 7.1: Pass usage relationship. Solid arrow means "uses", dashed arrow means "may use".

Each pass will now be explained in detail.

7.1 The OpenMPData pass

The *OpenMPData* pass is a container pass for immutable data structures used throughout the whole process. In particular the *OpenMPData* pass contains LLVM *Function* pointers representing OpenMP runtime functions. Through the LLVM API it is possible to interact with LLVM-IR compiled programs, modifying them or analyzing them.[27] Each part of an LLVM-IR program (like Basic-Blocks, Functions, Values and so on) has its respective LLVM *data structure* representing it in code. LLVM *Function* data structures, for example, represent LLVM-IR compiled functions [19]. LLVM objects are unique. This means that there will be no multiple LLVM objects representing the same LLVM-IR code part. So if there is an LLVM Function object representing a function named *func()* in the current Module, then can be assured that there will be only one instance of a Function object representing that function. This is quite advantageous for the programmer. It is possible to compare two LLVM objects just from their respective pointer values. If they represent the same LLVM code component then the pointer will be the same. This property is used in this project and by LLVM passes in general. In particular in this project it is needed to identify the presence of certain OpenMP runtime function calls inside the program. As explained in Section 5.1 we can identify regions of code containing OpenMP loop code by looking at certain specific runtime function calls that surround these regions.

These functions are:

1. **GOMP_parallel_start() :** This function identifies the beginning of a parallel code block.

2. **GOMP_parallel_end() :** This function identifies the end of a parallel code block.

3. **GOMP_loop_dynamic_start() :** This function identifies the beginning of an OpenMP loop code block.

4. **GOMP_loop_dynamic_next() :** This function identifies the beginning or the end of a loop iteration code. Loop code is usually wrapped by two basic-blocks containing calls to this function or between a call to a *GOMP_loop_dynamic_start()* function and this one.

5. **GOMP_loop_end() :** This function identifies the end of an OpenMP loop block.

6. **GOMP_parallel_loop_dynamic_start() :** This function is the combination of a call to the *GOMP_parallel_start()* function with a call to *GOMP_loop_dynamic_start()* in a loop block. It is used by the compiler in the case of combined "parallel" and "loop" constructs.

7. **GOMP_loop_end_nowait() :** The same as *GOMP_loop_end()*, but does not synchronize the loop threads between them.

This pass stores the pointers to the LLVM Function objects representing the functions described above and these will be used by the other passes to identify the presence of calls to these in the code.

7.2 The OpenMPNorm pass

The *OpenMPNorm* pass is a transformation pass with the purpose of normalizing LLVM-IR code to a common form so that the subsequent passes can make certain assumptions about the structure of the code they will receive as input. As stated in the previous section OpenMP code regions are usually contained between pairs of OpenMP runtime function calls. It is easier to identify these regions inside the code if these runtime calls are always at the beginning or at the end of a basic-block (just before the basic-block terminator). What the *OpenMPNorm* pass does is assuring that calls to the OpenMP runtime functions are always either at the beginning or at the end of a basic-block for easier retrieval.

An example of the input and the output of the *OpenMPNorm* pass can be seen in Figures 7.2 and 7.3. In this example is represented the LLVM-IR code (in graph form) of a function containing a *Parallel Block* that is identified by the pair of function calls highlighted in green: *GOMP_parallel_start()* and *GOMP_parallel_end()*. Between these two function calls is contained the code of the parallel block (a call to the compiler-generated function implementing the parallel block). The pass splits the basic block such that the first instruction of the basic-block is the call to the *GOMP_parallel_start()* function and the last instruction (the one before the terminator of the block) is the call to the *GOMP_parallel_end()* function. The entire *Parallel Block* gets contained in a single basic block exactly the size of the Parallel Block itself. It becomes then easy to identify a parallel block in the code. It is just necessary to iterate over all the basic blocks in a function and find a basic-block having a call to a *GOMP_parallel_start()* function as the first instruction and a call to a *GOMP_parallel_end()* function as its last. Without this pass it would have been necessary to iterate over all the instructions in a basic block to find the calls to the runtime defining the *Parallel Block*. The pass splits the blocks following certain rules:

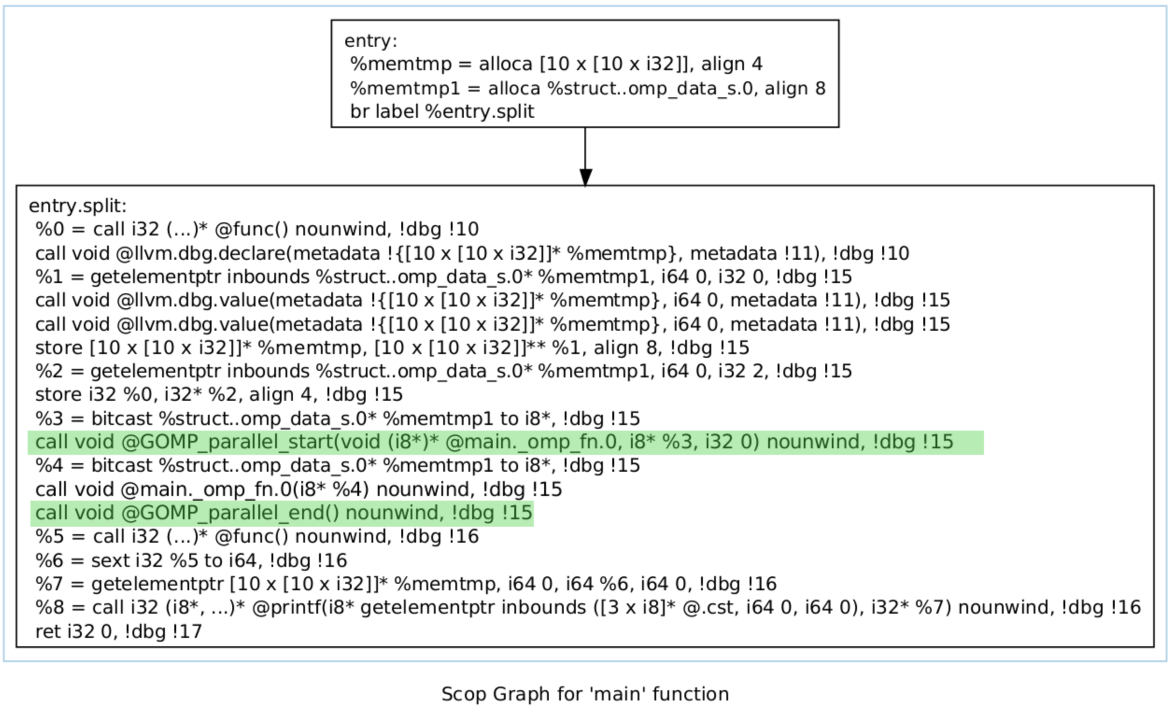
- When a call to *GOMP_loop_dynamic_next()*, *GOMP_loop_dynamic_start()*, *GOMP_loop_end()* or *GOMP_loop_end_nowait()* is encountered the pass splits the basic-block in which the call is found twice creating a basic-block containing only the call to the runtime function and the block terminator.
- If a call to either *GOMP_parallel_start()* or *GOMP_parallel_loop_dynamic_start()* is encountered then the basic-block is split only once and the pass tries to check for a call to a *GOMP_parallel_end()* or *GOMP_parallel_end_nowait()* function, in order to create a single basic-block containing an entire parallel block (as in the example of Figure 7.3).

The following pseudo code describes this pass behaviour:

```
bool SkipNextInstruction = false;
foreach (Function in CurrentModule) {
  foreach (BasicBlock in Function) {
    foreach (Instruction in BasicBlock) {
      if (SkipNextInstruction)
        continue;
      if (Instruction is a Call instruction) {

        CalledFunction = Function Called by Instruction;
        if (CalledFunction is GOMP_loop_dynamic_next or
            GOMP_loop_dynamic_start or GOMP_loop_end or GOMP_loop_end_nowait) {
          Split block at Instruction;
          Instruction = First instruction of the next Block;
          Split block at Instruction;
        } else if (CalledFunction is GOMP_parallel_start or
                   GOMP_parallel_loop_dynamic_start) {
          Split block at Instruction;
          SkipNextInstruction = true;
          break;

        } else if (CalledFunction is GOMP_parallel_end or
                   GOMP_parallel_end_nowait) {
          Split block at Instruction;
          break;
        }
      }
    }
  }
}
```



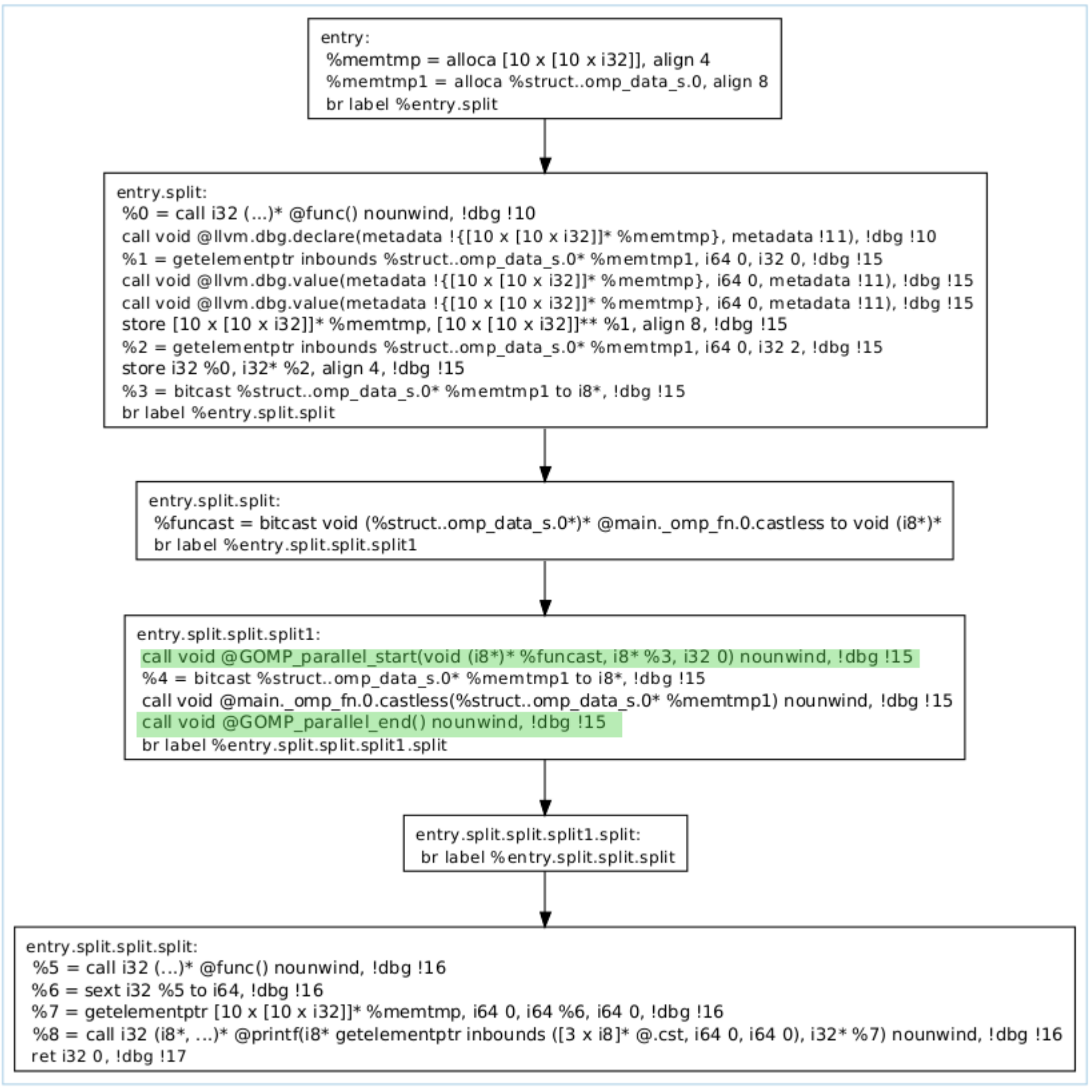
```
entry:
 %memtmp = alloca [10 x [10 x i32]], align 4
 %memtmp1 = alloca %struct..omp_data_s.0, align 8
 br label %entry.split
```

```
entry.split:
 %0 = call i32 (...)* @func() nounwind, !dbg !10
 call void @llvm.dbg.declare(metadata !{[10 x [10 x i32]]* %memtmp}, metadata !11), !dbg !10
 %1 = getelementptr inbounds %struct..omp_data_s.0* %memtmp1, i64 0, i32 0, !dbg !15
 call void @llvm.dbg.value(metadata !{[10 x [10 x i32]]* %memtmp}, i64 0, metadata !11), !dbg !15
 call void @llvm.dbg.value(metadata !{[10 x [10 x i32]]* %memtmp}, i64 0, metadata !11), !dbg !15
 store [10 x [10 x i32]]* %memtmp, [10 x [10 x i32]]** %1, align 8, !dbg !15
 %2 = getelementptr inbounds %struct..omp_data_s.0* %memtmp1, i64 0, i32 2, !dbg !15
 store i32 %0, i32* %2, align 4, !dbg !15
 %3 = bitcast %struct..omp_data_s.0* %memtmp1 to i8*, !dbg !15
 call void @GOMP_parallel_start(void (i8*)* @main._omp_fn.0, i8* %3, i32 0) nounwind, !dbg !15
 %4 = bitcast %struct..omp_data_s.0* %memtmp1 to i8*, !dbg !15
 call void @main._omp_fn.0(i8* %4) nounwind, !dbg !15
 call void @GOMP_parallel_end() nounwind, !dbg !15
 %5 = call i32 (...)* @func() nounwind, !dbg !16
 %6 = sext i32 %5 to i64, !dbg !16
 %7 = getelementptr [10 x [10 x i32]]* %memtmp, i64 0, i64 %6, i64 0, !dbg !16
 %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.cst, i64 0, i64 0), i32* %7) nounwind, !dbg !16
 ret i32 0, !dbg !17
```

Figure 7.2: A function containing OpenMP runtime calls before OpenMPNorm is run.

7.3 The OpenMPDetection pass

The *OpenMPDetection* pass is an analysis pass that is used to identify and feed to the subsequent passes information about OpenMP code regions. It expects code that has already been normalized by the *OpenMPNorm* pass. This pass identifies and stores two kinds of code regions: **Parallel Blocks** and **Loop Regions**. *Parallel Blocks* are LLVM-IR basic blocks that implement OpenMP parallel statements. *Parallel Blocks* begin with either a call to a *GOMP_parallel_start()* or *GOMP_parallel_loop_dynamic_start()* function and terminate with a call to a *GOMP_parallel_end()* or *GOMP_parallel_end_nowait()* function. An example of *Parallel Block* can be seen in Figure 7.3. *Loop Regions* are LLVM-IR basic blocks pairs. These two basic-blocks define a single entry/single exit region. The code in between these two basic-blocks is the code of an OpenMP loop. An example can be seen in Figure 7.4, where the first red block and the yellow block form a *Loop Region*.



```
entry:
 %memtmp = alloca [10 x [10 x i32]], align 4
 %memtmp1 = alloca %struct..omp_data_s.0, align 8
 br label %entry.split
```

```
entry.split:
 %0 = call i32 (...)* @func() nounwind, !dbg !10
 call void @llvm.dbg.declare(metadata !{[10 x [10 x i32]]* %memtmp}, metadata !11), !dbg !10
 %1 = getelementptr inbounds %struct..omp_data_s.0* %memtmp1, i64 0, i32 0, !dbg !15
 call void @llvm.dbg.value(metadata !{[10 x [10 x i32]]* %memtmp}, i64 0, metadata !11), !dbg !15
 call void @llvm.dbg.value(metadata !{[10 x [10 x i32]]* %memtmp}, i64 0, metadata !11), !dbg !15
 store [10 x [10 x i32]]* %memtmp, [10 x [10 x i32]]** %1, align 8, !dbg !15
 %2 = getelementptr inbounds %struct..omp_data_s.0* %memtmp1, i64 0, i32 2, !dbg !15
 store i32 %0, i32* %2, align 4, !dbg !15
 %3 = bitcast %struct..omp_data_s.0* %memtmp1 to i8*, !dbg !15
 br label %entry.split.split
```

```
entry.split.split:
 %funcast = bitcast void (%struct..omp_data_s.0*)* @main._omp_fn.0.castless to void (i8*)*
 br label %entry.split.split.split1
```

```
entry.split.split.split1:
 call void @GOMP_parallel_start(void (i8*)* %funcast, i8* %3, i32 0) nounwind, !dbg !15
 %4 = bitcast %struct..omp_data_s.0* %memtmp1 to i8*, !dbg !15
 call void @main._omp_fn.0.castless(%struct..omp_data_s.0* %memtmp1) nounwind, !dbg !15
 call void @GOMP_parallel_end() nounwind, !dbg !15
 br label %entry.split.split.split1.split
```

```
entry.split.split.split1.split:
 br label %entry.split.split.split
```

```
entry.split.split.split:
 %5 = call i32 (...)* @func() nounwind, !dbg !16
 %6 = sext i32 %5 to i64, !dbg !16
 %7 = getelementptr [10 x [10 x i32]]* %memtmp, i64 0, i64 %6, i64 0, !dbg !16
 %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([3 x i8]* @.cst, i64 0, i64 0), i32* %7) nounwind, !dbg !16
 ret i32 0, !dbg !17
```

Scop Graph for 'main' function

Figure 7.3: A function containing OpenMP runtime calls after OpenMPNorm is run.

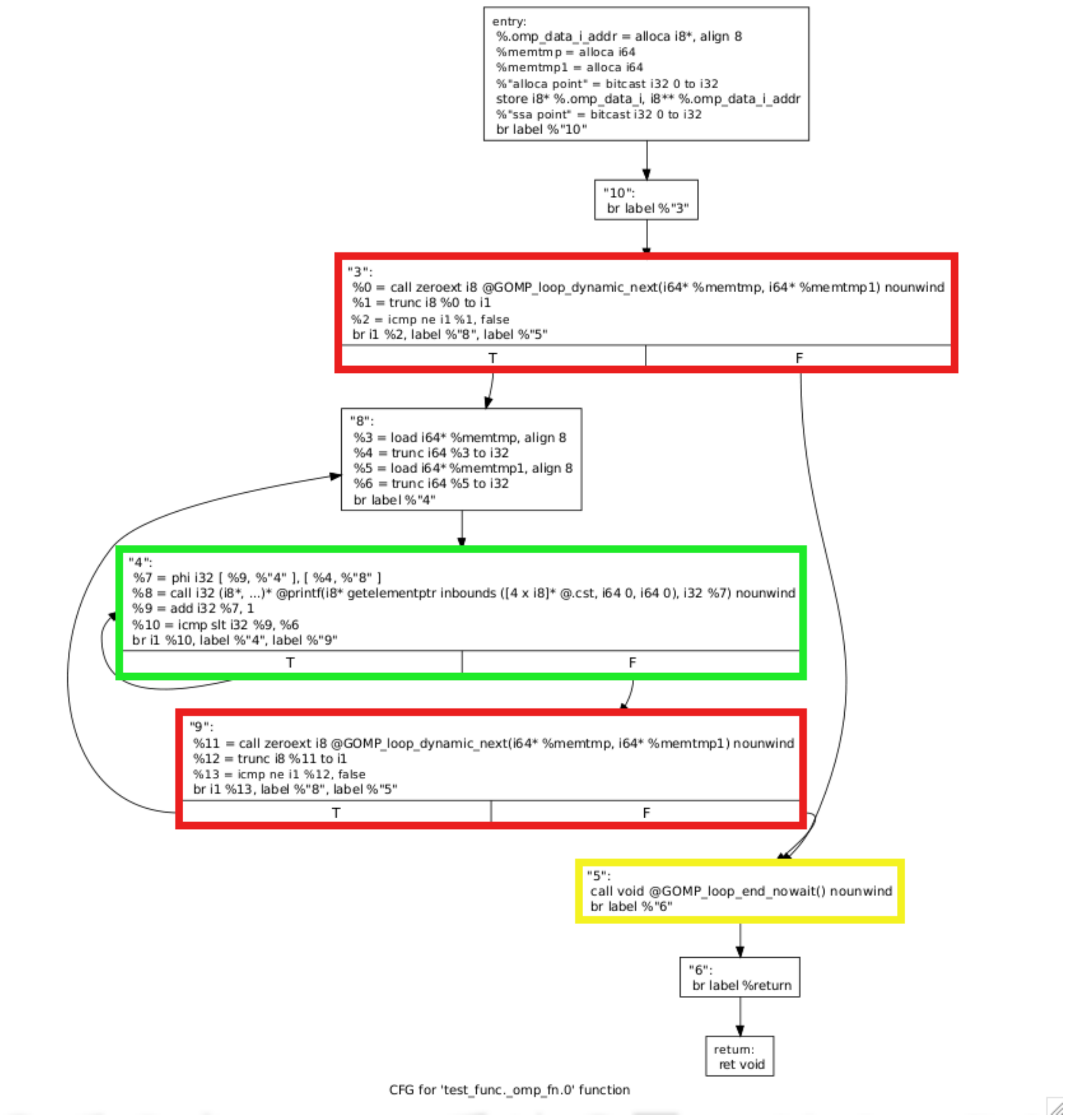
The process is simple and can is described by the following pseudo-code:

```
ParallelBlocks = Empty Vector of ParallelBlock objects;
LoopRegions = Empty Vector of LoopRegions objects;
foreach (Function in CurrentModule) {
  foreach (BasicBlock in Function in Depth-First order) {
    bool IsNotParallelLoopCombo = false;
    foreach (Instruction in BasicBlock) {

      if (Instruction is a Call instruction) {

        CalledFunction = Function Called by Instruction;
        if (CalledFunction is GOMP_parallel_start or
                   GOMP_parallel_loop_dynamic_start) {
          Add a new ParallelBlock(Function, BasicBlock) object to ParallelBlocks;

        } else if (CalledFunction is GOMP_loop_dynamic_start) {
          EndBlock = Final Block containing GOMP_loop_end or GOMP_loop_end_nowait;
          IsNotParallelLoopCombo = true;
          Add a new LoopRegion(BasicBlock, EndBlock) object to LoopRegions;
        } else if (!IsNotParallelLoopCombo &&
                  CalledFunction is GOMP_loop_dynamic_next) {
          EndBlock = Final Block containing GOMP_loop_end or GOMP_loop_end_nowait;
          Add a new LoopRegion(BasicBlock, EndBlock) object to LoopRegions;
          break;
        }
        }
      }
    }
  }
```



```
entry:
 %.omp_data_i_addr = alloca i8*, align 8
 %memtmp = alloca i64
 %memtmp1 = alloca i64
 %"alloca point" = bitcast i32 0 to i32
 store i8* %.omp_data_i, i8** %.omp_data_i_addr
 %"ssa point" = bitcast i32 0 to i32
 br label %"10"
```

```
"10":
 br label %"3"
```

```
"3":
 %0 = call zeroext i8 @GOMP_loop_dynamic_next(i64* %memtmp, i64* %memtmp1) nounwind
 %1 = trunc i8 %0 to i1
 %2 = icmp ne i1 %1, false
 br i1 %2, label %"8", label %"5"
```

```
"8":
 %3 = load i64* %memtmp, align 8
 %4 = trunc i64 %3 to i32
 %5 = load i64* %memtmp1, align 8
 %6 = trunc i64 %5 to i32
 br label %"4"
```

```
"4":
 %7 = phi i32 [ %9, %"4" ], [ %4, %"8" ]
 %8 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]* @.cst, i64 0, i64 0), i32 %7) nounwind
 %9 = add i32 %7, 1
 %10 = icmp slt i32 %9, %6
 br i1 %10, label %"4", label %"9"
```

```
"9":
 %11 = call zeroext i8 @GOMP_loop_dynamic_next(i64* %memtmp, i64* %memtmp1) nounwind
 %12 = trunc i8 %11 to i1
 %13 = icmp ne i1 %12, false
 br i1 %13, label %"8", label %"5"
```

```
"5":
 call void @GOMP_loop_end_nowait() nounwind
 br label %"6"
```

```
"6":
 br label %return
```

```
return:
 ret void
```

Figure 7.4: LLVM-IR code implementing the code of an OpenMP loop.

7.4 The OpenMPRegionTree pass

The *OpenMPDetection* pass identifies Parallel and Loop Regions, but does not provide any information about how they are structured together. The *OpenMPRegion-Tree* pass is an analysis pass that constructs a tree representing the parent-child relationship between Parallel Blocks and Loop Regions.

The relationship between Parallel Blocks and Loop Regions is such that the latter are usually contained inside the former, because to parallelize loops it is necessary to spawn threads through a parallel directive. Loop Regions not contained inside Parallel Blocks are not interesting, because the resulting loop would not be parallelized. OpenMP loops may be directly contained inside a parallel block , but they may also be put inside a function called inside a parallel block, like in the program shown in Figure 7.5 .

```c
int function() {
  int i;
#pragma omp for schedule(dynamic)
  for (i = 0; i < 10; ++i) {
    printf("TID: %d\", omp_get_thread_num());
  }
}

int main() {
#pragma omp parallel
  function();

  return 0;
}
```

Figure 7.5: An OpenMP loop inside a function.

While finding the relationship between a Parallel block and an OpenMP loop directly defined inside it is easy, the situation presented in Figure 7.5 is more complicated. In the former case we know that the OpenMP compiler-generated function for the parallel block contains the OpenMP loop implementation. In the latter case it is impossible to know in advance how many layers of function calls can be found between the parallel block and the loop implementation. It is needed, then, a way to track calls between functions and connect the OpenMP code structures through multiple function calls.

7.4.1 The Call Graph

A Call Graph is a directed graph where each node represents a function in the program, while each edge represents a caller-callee relationship between two functions. LLVM provides an Analysis pass that can be used to construct a Call Graph. An example of an OpenMP program and its associated Call Graph is shown in Figure 7.6 [20]. A Call Graph has a root node. If the program has an entry point (like a *main()* function) then the root node is the entry point of the program, otherwise, if the program does not have an entry point (like a library for example), the *external node* becomes the root node. The external node is a special node (not associated with any function) that represents functions not present in the module being analyzed by the LLVM Call Graph analysis pass. Functions in the module with external linkage (non static functions in C) may be called by functions outside the module itself. The *external node* models this possibility by having an edge pointing to each of the external functions in the module. There is another special node in the Call Graph. In Figure 7.6 it is the only node represented with an ellipse and it is used to represent unknown functions to the module. For example, there is no access to the source code for dynamically linked functions (like *C standard library* or OpenMP runtime functions) and because of that is not possible to know what functions these may call. This special node represents all the unknown functions that may be called by these non-analyzable library functions. The Call Graph may have cycles , representing recursive function calls. Each node in the call graph is represented by an instance of the CallGraphNode class [21]. This class keeps track of all the functions called by the routine the node represents thanks to a vector that references the children nodes. Together with the reference to the children nodes this class also stores a reference to the instruction that performs the call in the LLVM-IR code.

7.4.2 Building the Region Tree

The input from the previous pass (*OpenMPDetection*) is a list of all the *Loop Regions* and *Parallel Blocks* detected. The output of this pass is a graph arranging all these elements depending on their relationship in a tree. For the example in Figure 7.6 we would obtain a graph with two nodes: one node representing the Parallel Block in the *main()* function and one representing the loop in the *func()* function. The nodes would be arranged as in Figure 7.7. This pass stores the data about the structure of the OpenMP regions through a class called *OpenMPRegionNode* that represents the nodes of the region tree. The UML diagram for this class is shown in Figure 7.8.

```
int func() {
int i;
#pragma omp for schedule(dynamic)
  for (i = 0; i < 5; ++i) {
    int tid = omp_get_thread_num();
    printf("TID: %d\ n", tid);
    func2();
  }
}
void func2() {
  func();
}

int main() {
#pragma omp parallel
{
  func2();
}
  return 0;
}
```

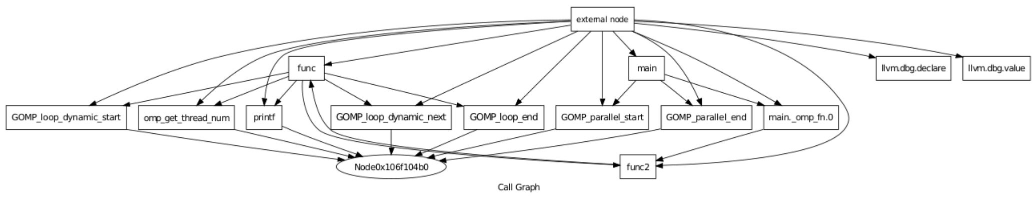


Figure 7.6: Example of an OpenMP C program with its Call Graph

The contents of this class are:

- The **Type** field, that is used to store the type of the region (either a Parallel block or a Loop Region) that the node represent.
- The **ContainingFunction** field, that is used to store a pointer to the LLVM Function object representing the function containing this region in the LLVM-IR code.
- The **BeginBlock** field, that is used to store a pointer to the LLVM BasicBlock object that represents the starting block of the region. Because Parallel regions are composed of only one block this is the same as *EndBlock* in case the Node represents a Parallel region.

- The **EndBlock** field, that is used to store a pointer to the LLVM BasicBlock object that represents the ending block of the region. Because Parallel regions are composed of only one block this is the same as *BeginBlock* in case the Node represents a Parallel region.

- The **ArgumentType** field, that is used to store a pointer to the LLVM Type object that represents the type of the data structure holding the arguments passed to the implementation function of a parallel directive. It is NULL if the region represents a Loop.

- The **Children** field, that is a C++ set holding pointers to other *OpenMPRegionNode* instances that are children of this node.

To build the graph the pass executes these steps:

1. Constructs the Nodes representing each OpenMP region we are interested into. For each of the parallel blocks and the loop pairs found by the *OpenMPDetection* a new *OpenMPRegionNode* is constructed. The Node is then stored in a map with a key equal to the pointer to the LLVM Function object of the function that contains the region.

2. The Call Graph is visited in a depth first manner, starting from the root node. The visit is performed through a recursive function , that takes as input the current Call Graph node being visited, a set of all the functions already visited by the routine (used to avoid cycles) and a pointer to the last OpenMPRegionNode object added to the region tree on this path of the call graph.

3. When visiting a new function it is checked if it has been already visited and in that case the visit function returns immediately (to avoid recursion cycles).

4. All the OpenMP regions contained in the function are collected from the map constructed at step 1 and each of their representing OpenMPRegionNode instances is added to the children set of the previous region node object passed as parameter to the visit function (see step 2).

5. For each function call in the current function the visit function is called recursively, to visit the next node of the call graph. If the call instruction performing the function call is contained inside an OpenMP region then a pointer to that region node is passed as a parameter to the next call to the visit function, otherwise , if the call instruction is not contained in an OpenMP region, then the region node parameter of the current call to the visit function is forwarded.

In step number 5 it is needed to determine if a certain function call is performed inside or outside an OpenMP region. If a function call is inside an OpenMP region, than this region will be the parent of all the OpenMPRegionNodes inside the called function. In the example of Figure 7.6 the parallel region in the *main()* function contains a function call to *func2()*. All the OpenMP regions contained in *func2()* (or in the functions it calls) will have to be added as children to the OpenMPRegionNode of the parallel region in *main()*. Determining if a call is contained into an OpenMP region or not is easy in the case of Parallel Regions. Parallel Regions are composed of only one basic block. From the CallGraphNode representing the function call we can obtain a pointer to the instruction that performs the function call inside the LLVM-IR code. If the call instruction is contained inside the only basic-block composing the Parallel Region then the function call is contained inside that OpenMP region. Only one function call (that is not an OpenMP runtime call) can be found in Parallel Regions. This is a call to the compiler-generated function that implements the parallel block. Loop Regions are usually composed of multiple basic blocks and of all these blocks only the entry block and the exit block of the region are known. Because of this the method used for Parallel Regions is not applicable to Loop Regions. It is needed to know if the basic-block containing the call instruction is positioned in between the entry block and the exit block of the Loop Region. Because a Loop Region is a single entry/single exit region then if the basic-block containing the call is *dominated* by the entry block and *post-dominated* by the exit block it is assured that it is contained in the region. This can be found out by using a Dominator Tree and a Post-Dominator Tree.

The Dominator Tree. *Dominance* is a concept of graph theory. A node *"a" dominates* another node *"b"* if every path that goes from the start node of the graph to *"b"* has to pass through *"a"* [22]. A related concept to Dominance is *Post-Dominance*. A node *"a" post-dominates* another node *"b"* if every path that goes from the node *"a"* to the exit node of the graph has to pass through *"b"*. *Dominance* and *Post-Dominance* are often used in compilers to do Control-Flow graph analysis. Dominator and Post-Dominator Trees are usually used to store information about dominance in a graph. In a Dominator tree the children of a node are the nodes it immediately dominates. When a Dominator Tree is used to analyze a control-flow graph the nodes of the tree are the basic-blocks of the program. The Dominator tree can then be used to determine dominance between basic-blocks of a program. LLVM provides analysis passes that construct Dominator and Post-Dominator Trees. These passes are called *DominatorTree* and *PostDominatorTree* [23] [24].

The pseudo-code for step 1 of the algorithm is:

```
void buildMaps() {
  foreach (ParallelRegion in OpenMPDetection region list) {
    ContainingFunction = Function containing ParallelRegion;
    ParallelRegionsMap[ContainingFunction] = new OpenMPRegionNode
                                         representing the parallel region;
  }
  foreach (LoopRegion in OpenMPDetection region list) {
    ContainingFunction = Function containing LoopRegion;
    LoopRegionsMap[ContainingFunction] = new OpenMPRegionNode
                                     representing the loop region;
  }
}
```

The pseudo-code for the visit function that explores the call-graph is:

```
void visit(CallGraphNode, Visited, CurrentRegionNode) {
  CurrentFunction = Function currently represented by CallGraphNode;
  ParallelRegionsVector = Vector of parallel regions contained in this function
                          obtained querying ParallelRegionsMap;
  LoopRegionsVector = Vector loop regions contained in this function
                      obtained querying ParallelRegionsMap;
  CallVisitedVector = Empty vector of CallGraphNodes;
  NodeSet = Pointer to the children set of CurrentRegionNode or
                  the RootNode if CurrentRegionNode is NULL;
  if (Visited contains CurrentFunction) {
    Add all the ParallelRegions and LoopRegions objects in this function to NodeSet
    return;
  }
  Add CurrentFunction to Visited;
  foreach (ParallelRegion in ParallelRegionsVector) {
    Node = OpenMPRegionNode representing ParallelRegion
    Add Node to NodesSet;
    foreach (CalledNode in CallGraphNode)
      Call visit(CalledNode, Visited, Node) if the call
      is contained in the ParallelRegion;
  }
  foreach (LoopRegion in LoopRegionsVector) {
    Node = OpenMPRegionNode representing LoopRegion
    Add Node to NodesSet;
    foreach (CalledNode in CallGraphNode)
      Call visit(CalledNode, Visited, Node) if the call is contained in a block
      that is Dominated by Node->BeginBlock and PostDominated by Node->EndBlock;
  }
  foreach (CalledNode in CallGraphNode)
    if (CalledNode function has no body (is external) ||
        CalledNode is contained into CallVisited)
      continue;
    visit(CalledNode, Visited, CurrentRegionNode);
  Remove CurrentFunction from Visited;
}
```

And finally the pseudo-code that uses the algorithms described in the *buildMap()* and *visit()* functions to build the Region tree:

```
RootNodeSet = Empty Set of region root nodes;

ParallelRegionsMap = Empty Multimap of OpenMPRegionNodes with Function as keys;
LoopRegionsMap = Empty Multimap of OpenMPRegionNodes with Function as keys;

void buildRegionTree() {
  buildMaps();
  buildTree();
}

void buildTree() {
  CallGraph = The Module CallGraph;
  CallGraphRoot = Root of CallGraph;

  VisitedFunctions = Empty set of functions;
  visit(CallGraphRoot, Visited, NULL);

}
```

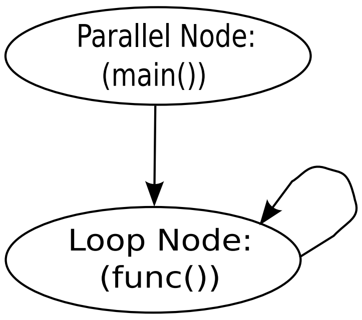


Figure 7.7: Result of an analysis with OpenMPRegionTree

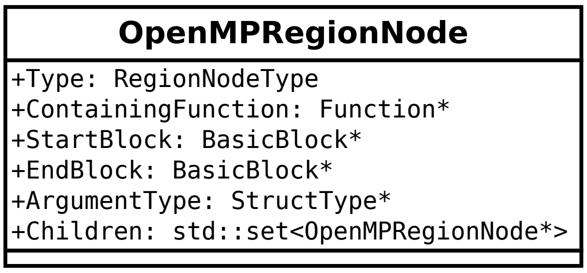

OpenMPRegionNode
+Type: RegionNodeType
+ContainingFunction: Function*
+StartBlock: BasicBlock*
+EndBlock: BasicBlock*
+ArgumentType: StructType*
+Children: std::set<OpenMPRegionNode*>

Figure 7.8: The OpenMPRegionNode class

7.5 The OMPNUMAIdentify pass

The *OMPNUMAIdentify* pass is the last pass run and the one that constructs the actual analysis of the loop. *OMPNUMAIdentify* is a SCoP pass. This is a special kind of pass introduced by Polly that is run once for every SCoP detected. This pass implements the the ideas explained in Section 5.3.

The steps executed by the pass are:

1. Validating the ScoP being analyzed. In order for a SCoP to be eligible for analysis it must be entirely contained in a Loop Region.
2. For each memory access contained in the SCoP extract its access function data.
3. Reconstruct the access functions for each array dimension.
4. Construct the ISL representation of the access functions for each array dimension.
5. Generate the result of the analysis from the dimensional access functions and the domain of the SCoP statement containing the memory access.

7.5.1 SCoP Validation

It is of interest analyzing only those SCoPs that are actually part of an OpenMP parallel loop. Using the tree constructed with the *OpenMPRegionTree* pass it is possible to check easily if a SCoP is contained in a parallel loop region. A SCoP is completely contained in a Parallel Loop region if all the basic blocks of the SCoP are dominated by the entry block and post-dominated by the exit block of an OpenMP loop region. If the loop region containing the SCoP has a parallel region as a parent in the RegionTree then it is a parallel loop region.

The pseudo code for SCoP validation is now presented:

```
RootNodeSet = OpenMPRegionNode set obtained
              by the OpenMPRegionTree pass;


bool validateSCoP(Scop) {
  foreach (Node in RootNodeSet) {
    Result = visitNode(Node, Scop);
    if (Result is true)
      return true;
  }
  return false;
}

bool visitNode(Node, Scop) {
  if (Node is Loop Region &&
      Node is contained in a Parallel Region) {
    IsValid = true;
    foreach (BasicBlock in Scop) {
      if (!Node->BeginBlock dominates BasicBlock ||
          !Node->EndBlock post-dominates BasicBlock) {
        IsValid = false;
        break;
      }
    }
    if (IsValid is true)
      return true;
  }

  foreach (Child in Node) {
    IsValid = visitNode(Node, Scop);
    if (IsValid is true)
      return true;
  }
  return false;
}
```

7.5.2 Array dimensional access functions determination

As explained in Section 5.3 Polly provides the access function for memory accesses as a single mono-dimensional affine function. If the original memory structure was a multidimensional array the information about the multiple dimensions of the array is lost. The targeted OpenMP runtime benefits from knowing how the multidimensional arrays are accessed in each dimension. It is possible to reconstruct the original array dimensional access functions by factorizing the access function provided by Polly using the array dimensions sizes.

This step can be divided in two parts:

1. Obtain the size of each Array dimension.
2. Factorize the memory access function provided by Polly to derive the dimensional array access functions.

Obtaining the array dimension sizes It is possible to derive the number of dimensions of an array and the size of each dimension directly from the LLVM-IR code through the LLVM API. In LLVM-IR code an array is defined through an *Alloca instruction*. The Alloca instruction reserves space for a data structure and returns a pointer to that space. Figure 7.9 shows an example usage of an alloca instruction that allocates a two dimensional array of 32-bit integers with each dimension of size 10. The alloca instruction contains information about the type of the data that is allocated by the instruction. The type is clearly visible in the example and is specified as *[10 x [10 x i32]]*. This type can be queried through the LLVM API.

The LLVM type system Types in LLVM are represented through an hierarchy of classes that has its root in the *Type* class. Each specific type is then represented by a specialized subclass of the *Type* class. There are sub-classes used to represent integral types, function types, structured data types, vector types and, finally, array types. Figure 7.10 shows the class hierarchy for the LLVM type system. The subclass representing array types is the *ArrayType* class. The *ArrayType* class provides a method called *getNumElements()* that returns the number of elements the array contains. The type of the elements contained in the array can be queried using the *getElementType()* function provided by the super-class *SequentialType*. Multidimensional arrays aren't directly supported by the *ArrayType* class, that is able to represent only one dimensional arrays. Multidimensional arrays can be represented by the LLVM type system treating them as *arrays of arrays*. A two dimensional array of integers is none other than an array of arrays of integers. Using this notion LLVM represents the type of a multidimensional array as an *ArrayType* that has as element type another *ArrayType*. The algorithm used to extract the array sizes for each dimension from an *ArrayType* object is (in pseudo-code):

```
ArrayDimensionSizes = Empty vector of integers;
Type = Type of the Array;

while (Type is an ArrayType) {
  Add Type->getNumElements() to ArrayDimensionSizes;
  Type = Type->getElementType();
}
```

Obtaining the type of the ArrayType object of the array Polly stores the load and store instructions used to read or write memory for each memory access it detects in the SCoP. This instruction refers to the array allocated through the *Alloca instruction* from which we can obtain the ArrayType object of representing the accessed array type.

```
%memtmp = alloca [10 x [10 x i32]], align 4
```

Figure 7.9: Example of usage of an alloca instruction

Extracting the dimensional memory access functions Polly provides to the pass the memory access function as an ISL map object. The affine function is of the form:

$$f(i0, i1, ...iX, p0, ...pX) = a \star i0 + b \star i1 + ... + c \star iX + k \star p0 + l \star pX + C$$

where iX are the value of the iterators of the loops surrounding the access, pX are the value of the parameters of the SCoP statement and a,b,c,k,C are constant integer values. As explained in Section 5.3.2 multiple dimension access functions are fused into a single one dimensional access function of the form above. The array access in Figure 7.11 would be translated to this one-dimensional access function

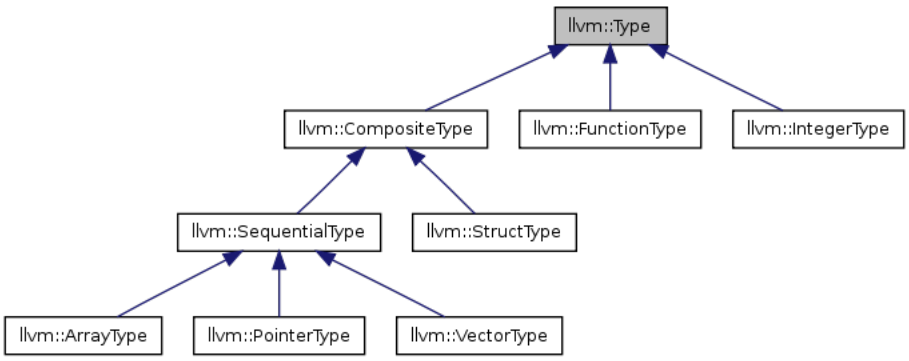


Figure 7.10: The LLVM Type system class heirarchy.

(the example is written in C):

$$f(i,j) = 15 \star i + 1 \star j$$

Multidimensional array elements are placed in memory at contiguous addresses. Different languages may dispose the elements in different ways in memory. For example C and C++ use a row-major order for arrays [25]. This means that each row of the array is disposed contiguously in memory. Other languages , like Fortran and Matlab use a column-major order for arrays. In the example iterator *"i"* is multiplied by the constant 15 because, being C a row-major language increasing by one the value of iterator *"i"* skips a whole row of dimension 15 . If the size of each array dimension is available it is possible to reconstruct each dimension access function by applying factorization to the mono-dimensional affine relation. Factorization also helps in identifying which iterators are involved in which array dimension. For the example of Figure 7.11 it is need to find out, from the access function above, that iterator *"i"* is accessed from the second dimension of the array and iterator *"j"* from the first one. If an iterator is accessed from an array dimension then we are sure that **its coefficient is a multiple to all the array dimensions sizes that come before that dimension**. In the case of the access in Figure 7.11 iterator *"i"* is multiple to the size of the first array dimension (that is 15), while the coefficient of iterator *"j"* is multiple to no other dimension size , because it is used in the rightmost dimension (so no other dimension are before it).

The factorization process can be outlined by these steps:

1. Compute the product between all the array dimensions sizes with the exception of the most significant dimension (the left most dimension in the array definition

in the case of row-major languages).

2. For each array dimension (from the most-significant to to the least-significant) it is needed to divide each of the loop iterator and parameter coefficients in the mono-dimensional access function by the value computed at step 1.

3. At each step the coefficient of the iterator/parameter for that array dimension is the result of the division between the value computed at step 1 and the coefficient itself. The coefficients values of the mono-dimensional access function are updated as the modulo between themselves and the value computed at step 1.

4. After the computation for a certain dimension is done the value obtained at step 1 has to be updated dividing it by the size of the the array dimension that has just being considered.

The pseudo-code for this process is the following:

```
IteratorsCoefficients[] = Array containing the coefficients
                          for the memory access function iterators;
ParametersCoefficients[] = Array containing the coefficients
                          for the memory access function parameters;
Constant = Integer representing the constant of the affine access function;
IteratorsForDimension[][] = Array containing the coefficients of
                              iterators for each array dimension;
ParametersForDimension[][] = Array containing the coefficients of
                               parameters for each array dimension;
ConstantForDimension[] = Array containing the constants for each
                           array dimension;
ArrayDimensionsSizesNoLast = Vector of integers containing the size in
                               elements of each dimension with exception of
                               the last dimension (the most external one);
ArraySize = 1;
foreach (Size in ArrayDimensionsSizesNoLast)
  ArraySize = ArraySize*Size;
//Starting from the most significant dimension to the least significant ...
for (int i = NumberOfArrayDimension-1; i >= 0; --i) {
  for (int j = 0; j < NumberOfLoopIterators; ++j)
    if (IteratorsCoefficients[j] >= ArraySize) {
      IteratorsForDimension[i][j] = IteratorsCoefficients[j] / ArraySize;
      IteratorsCoefficients[j] = IteratorsCoefficients[j] % ArraySize;
    }
  for (int j = 0; j < NumberOfLoopParameters; ++j)
    if (IteratorsCoefficients[j] >= ArraySize) {
      ParametersForDimension[i][j] = ParametersCoefficients[j] / ArraySize;
      ParametersCoefficients[j] = ParametersCoefficients[j] % ArraySize;
    }
  if (Constant >= ArraySize) {
    ConstantForDimension[i] = Constant / ArraySize;
    Constant = Constant % ArraySize;
  }
  ArraySize = ArraySize / Size of array dimension i;
}
```

In Figure 7.12 there is a more complex example of array access in C. It is an array with four dimensions and the array is accessed through the three iterators *i,j,k*. Iterator *"i"* is used twice. Even in this situation (with iterators used multiple times) the algorithm behaves correctly. *ArraySize* in this case would be:

$$15 \star 20 \star 25 = 7500$$

and the mono-dimensional access function:

$$f(i, j, k) = 7500 \star k + 500 \star j + 26 \star i$$

After executing the the first iteration of the above algorithm over this access function, the access function for the most-significant dimension is returned as result:

$$D_4(i, j, k) = 1 \star k + 0 \star j + 0 \star i$$

This is obtained by dividing the coefficient of each loop iterator by 7500, that is the value of *ArraySize* at the first iteration. The computed access function for dimension 4 is correct , as memory is accessed through only iterator *"k"*. After the first iteration each coefficient in the access function is updated with the modulo between the old value and the value of *ArraySize*. The updated mono-dimensional function then becomes:

$$f(i, j, k) = 0 \star k + 500 \star j + 26 \star i$$

And in the end *ArraySize* is updated dividing it by the size of the current dimension being considered which is 15. This process is repeated for each dimension and at the end the result for each array dimension is the correct memory access function for that specific dimension.

```
int A[10][15], i, j;

//...

A[i][j] = 0;
```

Figure 7.11: Example of C code accessing a bi-dimensional array.

```
int A[10][15][20][25], i, j,k;

//...

A[k][j][i][i] = 0;
```

Figure 7.12: Example of C code accessing a 4-dimensional array.

7.5.3 Result computation

After the previous step the memory access functions for each array dimension are computed. These will now be used to compute the result of the analysis.

Access function coefficients to ISL conversion The ISL library is used to compute the result of the analysis. To make use of ISL it is needed to convert the computed access functions coefficients to an ISL data structure that can be used with ISL functions to make the actual computations. The data structure to which the computed memory access functions will be converted into is the *isl_aff* type . The *isl_aff* type represents quasi-affine expressions in ISL. The process consists into:

1. Creating one empty ISL affine object for each iterator coefficient.
2. Set the value of the of the ISL affine objects to the constant value representing the coefficient that they represent
3. Store the result into a vector.

One ISL affine object is constructed for each coefficient of the loop iterators for each dimension of the array. If the array has 2 dimensions and it is nested in a loop of depth 3 the number of ISL objects constructed is 6. While the ISL representation of the coefficients is constructed it is also checked if the iterator of the most external loop iterator has a non-zero coefficient in that access function. In Section 5.2 is mentioned the difference between *Slices* and *Ranges*. What differentiates a *Slice* from a *Range* is that a *Slice* is obtained when the dimension is accessed through the iterator of the most external loop, while a *Range* is obtained otherwise. The information about the presence of the external loop iterator in the dimensional access function is used to decide if the result will be a *Range* or a *Slice*.

The pseudo-code for this step is now presented:

```
AffineIteratorCoefficients[][] = 2-dimensional array of the affine function
                                 coefficients in ISL form;

foreach (Dimension of the Array) {
  AffineFunction = Empty ISL object representing an Affine Function;
  HasExternalLoopIterator = false;
  foreach (Iterator of the loop) {
    if (Iterator of the external loop && IteratorCoefficient != 0)
      HasExternalLoopIterator = true;
    Set AffineIteratorCoefficients[Dimension][Iterator]
    as a constant affine function of the value of IteratorCoefficient;
  }

  Flag this dimension with HasExternalLoopIterator;
}
```

After this process the iterator coefficients of each access function derived at the previous step is now expressed in ISL form. For example, if one of the access functions have the form:

$$f(i,j) = 10 \star i + j + 5$$

where *"i"* and *"j"* are two loop iterators (not parameters) the algorithm will produce two ISL affine objects one containing the value 10 and the other containing the value 1.

Iterator bounds computation The next step consists in computing the maximum and minimum values of the memory access functions. A prerequisite in doing so is computing the upper and lower bounds of each loop iterator in the SCoP statement we are considering. Each memory access the pass analyzes is contained into a SCoP statement. As explained in Section 3.1 each SCoP statement has its own domain that specifies the value of the loop iterators for which the statement is executed. The memory access relations are functions of the loop iterators, by finding the upper bounds and lower bounds of the loop iterators, with respect to the domain of the statement, it is possible , by substituting these bounds into the function expression, to find the upper and lower bounds of the memory access relation. For the computation of the iterator bounds ISL provides functions that compute the upper and lower bound of the variables that live in a domain. These are the *isl_set_dim_max()* and *isl_set_dim_min()* functions, that can be used to determine the upper and lower bounds of dimensions in an ISL set object (SCoP statement domains are expressed as ISL sets by Polly). These two functions return an *isl_pw_aff* object. This data structure represents piecewise quasi-affine functions in ISL that are aggregations of quasi-affine functions. The *isl_pw_aff* data structure can be seen as an aggregation of *isl_aff* objects. These functions return objects that don't live in the same space as the domain set (they don't have the same number of iteration variables or parameters), so it is needed to correct them by adding the missing dimensions. The number of dimensions must be comparable with the dimensions of the domain or it won't be possible to use these objects in the next steps (ISL supports operations only between affine functions with a comparable number of dimensions and parameters).

The pseudo-code for this process is:

```
BoundsVector = Vector of Pairs of affine functions;

foreach (Iterator in SCoPStatementDomain) {
  UpperBound = Compute Upper bound of Iterator through ISL;
  Fix the dimensions of UpperBound to match with those of SCoPStatementDomain;
  LowerBound = Compute Lower bound of Iterator through ISL;
  Fix the dimensions of LowerBound to match with those of SCoPStatementDomain;

  Add the UpperBound/LowerBound pair to BoundsVector;
}
```

Access functions bound computation With the loop iterator bounds computed at the previous step it is possible to evaluate the bounds of the memory access functions. In Section 7.5.2 memory access functions for each of the array dimensions have been constructed. The available information about memory access functions at this stage are:

1. The coefficients of the iterators.
2. The upper bounds and lower bounds of the iterators in the SCoP statement domain.

To compute the upper and lower bounds we substitute the bounds of the loop iterators computed at the last step (with the exception of the bounds of the external loop iterator, that are considered as zero) inside the access function expression. The external loop iterator bounds values are considered as zero, because the objective of the analysis is to determine the amount of elements each external loop iteration visits (as explained in Section 5.2). By taking into account also the bounds of the external loop iterator in the computation the resulting upper and lower bounds of the access function would refer to the whole loop execution instead of only one iteration of the external loop. The example in Figure 7.13 has the following mono-dimensional access function:

$$f(i,j) = 100 \star i + 100 \star k + j + q$$

while the access function for dimension 1 is:

$$D_1(i,j) = 0 \star i + 0 \star k + j + q$$

and the access function for dimension 2 is:

$$D_2(i,j) = i + k + 0 \star j + 0 \star q$$

After the lower and upper bound computation for the loop iterators the result would be for loop iterator *"i"*:

$$UB_i = 90$$

$$LB_i = 0$$

for loop iterator *"j"*:

$$UB_j = 90$$

$$LB_j = 0$$

for loop iterator *"k"*:

$$UB_k = 9$$

$$LB_k = 0$$

and for loop iterator *"q"*:

$$UB_q = 9$$

$$LB_q = 0$$

Substituting these bounds inside the dimensional access functions gives for dimension 1:

$$UB_1 = 0 \star 0 + 0 \star 9 + 90 + 9 = 99$$

$$LB_1 = 0 \star 0 + 0 \star 0 + 0 + 0 = 0$$

and for dimension 2:

$$UB_2 = 0 + 9 + 0 \star 99 + 0 \star 9 = 9$$

$$LB_2 = 0 + 0 + 0 \star 0 + 0 \star 0 = 0$$

Here is the pseudo code for this step:

```
UpperBounds = Vector containing the upper bound affine
              expressions for each array dimension;
LowerBounds = Vector containing the lower bound affine
              expressions for each array dimension;

foreach (Dimension of the Array) {

  LowerBound = Zero affine expression;
  UpperBound = Zero affine expression;

  //The add and multiply operations here are
  //Operations performed on affine functions
  foreach (Iterator of the Loop nest) {
    LowerBound = LowerBound +
                 Coefficients[Iterator] * IteratorLowerBound[Iterator];
    UpperBound = UpperBound +
                 Coefficients[Iterator] * IteratorUpperBound[Iterator];
  }

  Append LowerBound To LowerBounds;
  Append UpperBound To UpperBounds;

}
```

Slices and Ranges computation With the data derived until this point it is possible to determine the *Slices* and *Ranges* for each dimension. The definitions of Slices and Ranges have been explained in Section 5.2. Before computing and storing the results for the Slices and the Ranges there's still another missing part that needs to be added to the access function bounds. The bound data still misses information about parameters and constants of the access functions. Constants and parameters do not change during loop iterations, and are called *loop invariants*. Considering loop invariants during bound computation is superfluous and complicates the process as these can be added to the Slice and Range results just before storing them. Loop

```
#define VBLOCK 10
#define HBLOCK 10
int A[100][100], i, j, q, k;

#pragma omp parallel
{
#pragma omp for private(i,j,k,q) schedule (dynamic)
for (i=0; i < 100; i+=VBLOCK)
  for (j=0; j < 100; j+=HBLOCK)
    for (k=0; k < VBLOCK; ++k)
      for (q=0; q < HBLOCK; ++q)
        A[i+k][j+q] = j;

}
```

Figure 7.13: Example of an OpenMP C loop accessing a 2-dimensional in blocks array.

invariants are added to the affine expressions of the bounds by constructing an affine expression containing only the values of the loop invariants and adding this expression to the lower and upper bound expressions of the access functions. In the example of Figure 7.14 can be seen a memory access with a parameter and a constant value added in the first dimension. The access function for dimension 1 is:

$$D_1 = 0 \star i + j + n + 5$$

where "i" and "j" are loop iterators, while "n" is a parameter of the SCoP and 5 is a constant. The parameter "n" and the constant 5 are loop invariants. After the upper and lower bounds computation for dimension 1 the result would be:

$$UB_1 = UB_j = 14$$

$$LB_1 = LB_j = 0$$

The loop invariants affine expression for dimension 1 is:

$$LI = n + 5$$

Adding this expression to the bounds would give these final bounds:

$$FinalUB_1 = UB_1 + LI = 14 + n + 5$$

$$FinalLB_1 = LB_1 + LI = 0 + n + 5$$

These are the true final results for the upper and lower bounds of dimension number 1. These bounds can now be used directly to store the Slices and Ranges values for each dimension. For the Ranges the final values of the upper and lower bounds are directly stored in a data structure as the result of the analysis. Slice computation requires one extra step, because the result for the Slice size is the difference between the upper and the lower bounds values.

```
#define VBLOCK 10
#define HBLOCK 10
int A[10][15], i, j, n;

n = foo();

#pragma omp parallel
{
#pragma omp for private(i,j) firstprivate(n) schedule (dynamic)
for (i=0; i < 10; ++i)
  for (j=0; j < 15; ++j)
    A[i][j+n+5] = j;

}
```

Figure 7.14: Example of an OpenMP C loop accessing a 2-dimensional array with a parameter and a constant.

The result of the analysis of a memory access is stored inside a class called *MemoryAccessResult*. *MemoryAccessResult* is a container class that stores two objects:

- A pointer to the LLVM Instruction object that does the memory access.
- A vector containing the result for each dimension of the array (either a Slice result or a Range result)

The UML diagram representing the *MemoryAccessResult* class is showed in Figure 7.15. As shown in the diagram there are methods implemented to get or set the information stored in the class. The vector (a SmallVector, that is an optimized implementation of the C++ vector class inside LLVM) contains pointers to instances of the *AnalysisResult* class. *AnalysisResult* is the base class for two classes used to store the actual results. These sub-classes are called *SliceResult* and *RangeResult*. *RangeResult* is used to store the data about a Range of an array dimension, while *SliceResult* stores the data for Slices. The UML diagram for these classes can be seen in Figure 7.16. Slices are represented with a single value, while Ranges are a pair of values representing the lower and upper bound of the range. These classes store the values through the use of another class called *BoundExpression*. *SliceResult* implements a *getSliceSize()* method to get the Slice size expression contained in the class, while *RangeResult* implements two methods, *getUpperBound()* and *getLowerBound()*, that return respectively the Upper and Lower bound expressions of the range. The format of values stored by a *BoundExpression* class is:

$$Expr = C + a \star p0 + b \star p1 + ... + k \star pN$$

where C is a constant value and *p0* to *pN* are the parameters of the SCoP statement in which the memory access that has been analyzed lives into. *BoundExpression* stores the constant value in an integer variable, while the parameters are stored in a vector of *Parameter* objects. The *Parameter* class is a simple container class that stores the coefficient and the index of a SCoP parameter. The UML diagram for these two classes is represented in Figure 7.17. The *BoundExpression* class provides methods to inspect the value it represents:

- **isConstant()** is a method that can be used to know if the value represented by the class is a completely constant value or it involves any parameter.
- **begin()** is a method that is used to obtain the begin iterator to the list of parameters in the expression.
- **end()** is a method that is used to obtain the iterator to the end of the list of parameters in the expression.
- **printToStr()** is a method used to obtain a string representation of the expression.

Each instance of the *MemoryAccessResult* class is then stored inside a vector contained in the *OMPNUMAIdentify* pass instance. This vector can be retrieved by other analysis or transformation passes to be used to instrument code with the necessary OpenMP runtime calls that inform the runtime of the data access patterns of the program.

This is the pseudo-code for this last part of the pass:

```
UpperBoundsVector = Vector of affine functions representing
                    the computed upper bounds;
LowerBoundsVector = Vector of affine functions representing
                    the computed lower bounds;
MemoryAccessResult = Object storing the results of the analysis;

foreach (Dimension of the Array) {
  LoopInvariantExpression = The affine Loop invariant
                            expression for this Dimension;
  UpperBound = Upper bound from UpperBoundsVector for Dimension;
  UpperBound = UpperBound + LoopInvariantExpression;
  LowerBound = Lower bound from LowerBoundsVector for Dimension;
  LowerBound = LowerBound + LoopInvariantExpression;
  if (Dimension is flagged with HasExternalLoopIterator) {
    //It is a Slice
    Result = UpperBound - LowerBound;
    Store Result into MemoryAccessResult for Dimension;
  } else {
    //It is a Range
    UBLBPair = A pair object containing the UpperBound
               and LowerBound expressions;
    Store UBLBPair into MemoryAccessResult for Dimension;
  }
}
```

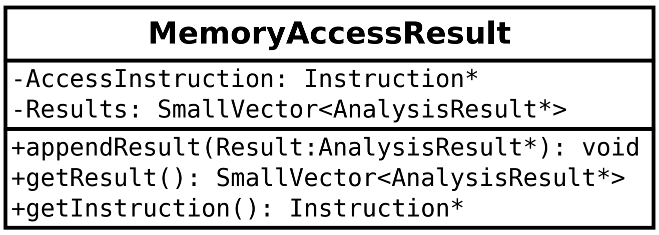

MemoryAccessResult
-AccessInstruction: Instruction*
-Results: SmallVector<AnalysisResult*>
+appendResult(Result:AnalysisResult*): void
+getResult(): SmallVector<AnalysisResult*>
+getInstruction(): Instruction*

Figure 7.15: The UML diagram for the MemoryAccessResult class.

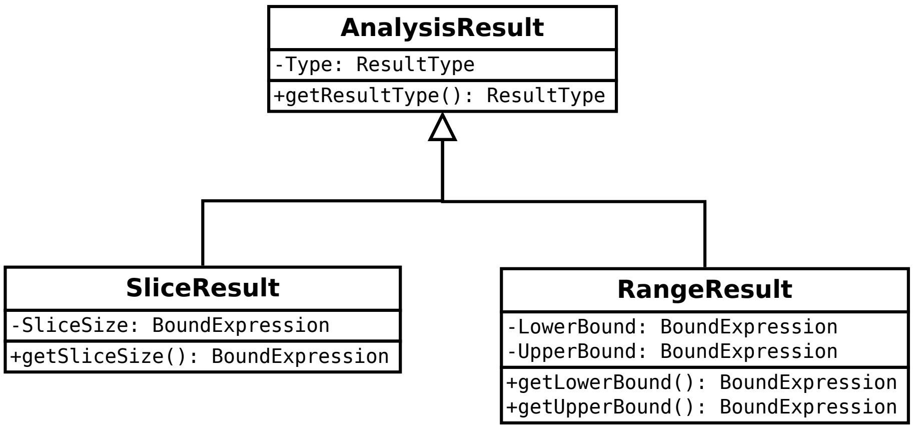

AnalysisResult
-Type: ResultType
+getResultType(): ResultType
SliceResult
-SliceSize: BoundExpression
+getSliceSize(): BoundExpression
RangeResult
-LowerBound: BoundExpression
-UpperBound: BoundExpression
+getLowerBound(): BoundExpression
+getUpperBound(): BoundExpression

Figure 7.16: The UML diagram for the AnalysisResult class and its sub-classes.

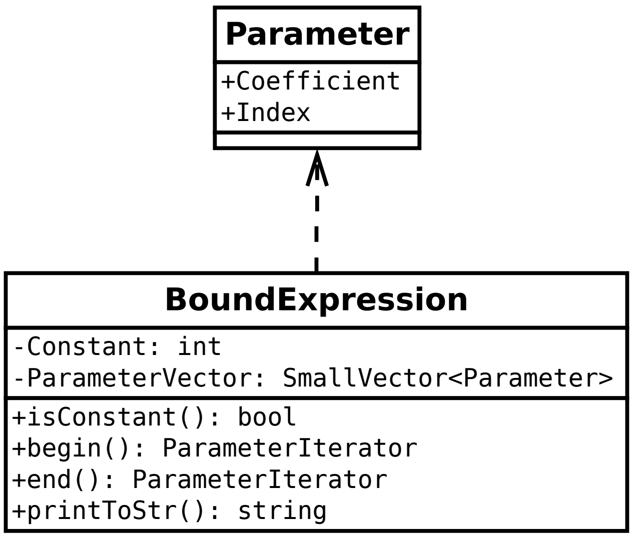


Figure 7.17: The UML diagram for the BoundExpression and Parameter classes.

Chapter 8

Experimental Results

In this chapter will be presented some experimental results. The *OMPNUMAIdentify* pass will be run on the code implementing some OpenMP parallelized algorithms. The aim of this chapter is to show how the analysis works on some test programs and known real world algorithms. The analysis will return the size of the Slices or the Ranges of the arrays accessed in the parallel loops of the analyzed algorithms. To execute the pass as a standalone application it is used a self-developed tool that runs the passes outside of the LLVM compiling pipeline directly on some LLVM-IR Assembly code (denoted by the *.ll* file extension).

8.1 Code preparation

The code in these examples is compiled into LLVM assembly code using the GCC compiler [1] with the addition of the DragonEgg plugin [28], that gives GCC the ability to emit LLVM code or use LLVM as a backend. The command used to compile C code into LLVM assembly through GCC+DragonEgg is the following one:

```
$ gcc -fopenmp -S -c -O0 -o $OUTPUT.ll $INPUT.c -fplugin=$PATHTODRAGONEGG/dragonegg.so -flto
```

The *-fopenmp* flag is used to enable the GCC OpenMP frontend support, the *-S* flag is used to tell GCC to emit assembly code instead of producing object code and the *-c* flag is used to make GCC skip the linking phase. *-O0* is given to disable all optimization , because we want to run optimizations in a later phase. The *-fplugin* flag is used to load the DragonEgg plugin into GCC, while the *-flto* flag tells the

[1]http://gcc.gnu.org/

plugin to generate LLVM assembly code instead of LLVM object code (that is not human readable).

Now the LLVM assembly code has been compiled into a file and can be passed through the LLVM optimizer. The optimizer can be run as a standalone program called *opt*. The *opt* application outputs a version of the LLVM program code modified by the optimizations specified to the *opt* command line. This stage of optimization is used to prepare the code for Polly analysis. Polly requires the code to be normalized into a specific form for analysis and to transform the code into this form some transformations passes have to be run before handing the code to Polly. These transformation passes are run automatically by the *-O3* level of optimization of *opt* , but running *-O3* optimizations changes the code shape pretty heavily, making it difficult to show similarities between the original C code and the compiled LLVM code so, for the sake of clarity, the required LLVM optimizations for Polly analysis will be run manually through *opt*, avoiding those *-O3* optimizations that heavily change the code shape. The *opt* command used to generate the normalized LLVM code for Polly is:

```
$ opt -S -load $PATH_TO_POLLY_LIB/LLVMPolly.dylib --debug-pass=Structure -no-aa -targetlibinfo \
   -tbaa -basicaa -preverify -domtree -verify -mem2reg -instcombine -simplifycfg -tailcallelim \
   -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -instcombine \
   -scalar-evolution -loop-simplify -lcssa -indvars -polly-prepare -postdomtree -domfrontier \
   -regions -polly-region-simplify -scalar-evolution -loop-simplify -lcssa -indvars -postdomtree \
   -domfrontier -regions  -loop-simplify -indvars  -enable-iv-rewrite -dce INPUT.ll > OUTPUT.ll
```

The command specifies a lot of LLVM passes to optimize the code. Most of them are standard simplification passes for loops and the CFG (*-loop-simplify* and *-simplifycfg*), while some others normalize the code into forms more easily analyzable (*-enable-if-rewrite* and *-polly-prepare*). The meaning of most these passes can be found on the LLVM website [1].

After *opt* has been run with the command above the *OUTPUT.ll* file is ready to be passed to the analysis tool.

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

8.2 Analysis results

8.2.1 Test application 1

The first program presented as a test is a simple OpenMP loop accessing a two dimensional array linearly. The code and the analysis results for the test program are presented in Figure 8.1. The external loop is driven by iterator *"i"* and the the most internal loop is driven by iterator *"j"*. The array accessed at *SCoP statement 1* is a two dimensional array with both dimensions of size 10. The whole array is accessed linearly row by row. The result of the analysis shows a Slice found for *Dimension 0* (the left-most dimension) of size 1 and a Range found for *Dimension 1* that spans between the constant values 0 and 9. The access can be be seen in Figure 8.2, where the strong-green elements are those accessed after the first iteration of the external loop. That access pattern is replicated through the rest of *Dimension 0* to cover the entire array (as shown in light-green).

```
int main() {

  int A[10][10], i,j;

#pragma omp parallel
{
i = 0;
//SCoP 1 begins here
#pragma omp for private(i,j) schedule(dynamic)
  for (i = 0; i < 10; ++i) {
    for (j = 0; j < 10; ++j) {
      A[i][j] = 0; //SCoP statement 1
    }
  }
}
  return 0;
}

Analysis Result:

1) Found memory access at line: 11
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 9
```

Figure 8.1: Code for Analysis test application number 1.

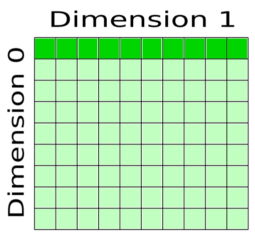


Figure 8.2: Access pattern for test application 1.

8.2.2 Test application 2

Test program number 2 is an OpenMP loop nest accessing a two dimensional array in blocks of size 4. The code for this program and the analysis result is presented in Figure 8.3. The array is accessed in 2x2 blocks skipping the first two columns. The program introduces a parameter for the SCoP. The parameter is the variable *"h"*, that is constant throughout the execution of the only SCoP in the program. The access pattern for this memory access is shown in Figure 8.4 (with respect to one iteration of the external loop). The access has a Slice for *Dimension 0* of size 2 , while *Dimension 1* is a Range with parametric bounds. The lower bound is equal to the parameter called *p2*, while the upper bound is *p2 + 7*. The parameter *p2* is none other than the variable *"h"* in the source code. The result is correct, because when all the iterators of the loops are zero we have the lower bound for the access function of *Dimension 1*, which leave *"h"* as the only addend in the equation if *"f"* and *"k"* are substituted with 0. When iterators *"f"* and *"k"* reach their maximum value we have the upper bound for *Dimension 1* access function. The sum between the two iterators at their maximum value gives 7 as result, which brings to the final result of *7 + p2* for the upper bound value.

8.2.3 LU Decomposition Algorithm

The third test program is a C implementation of the dense LU Decomposition algorithm[26][30], run on a 100x100 matrix of floating point values (implemented as a two dimensional array). The code and result for the algorithm is presented in Figure 8.5. This algorithm has many memory accesses (7 accesses in total). Each access is simple, with only one iterator or parameter involving each array dimension. Most of the accesses seen here are similar to the ones analyzed in the two already proposed test programs. Access number 6 and number 3 have some differences worth an explanation. Access number 6 corresponds to the access *LU[i][i]* that takes place

```
int main()
{
  int a[10][10];
  int i,j,k,f,h = 2;
//SCoP 1 begins here
#pragma omp parallel for private(i,j,k,f) firstprivate(h) schedule(dynamic)
  for (i=1; i < 5; ++i)
    for (f=0; f < 4; ++f)
      for (j=0; j < 2; ++j)
        for (k=0; k < 2; ++k)
          A[2*i+j][2*f+k+h] = 0; //SCoP Statement 1

        return 0;
}

Analysis Result:

1) Found memory access at line: 10
        Dimension 0 Slice of size: 2
        Dimension 1 Range with LB: p2  UB: 7 + p2
```

Figure 8.3: Code for Analysis test application number 2.

in *SCoP statement 1*. It is a constant memory access (accessing always to the same element). Parameter *p1* is the variable *"i"* in SCoP 1. Access 6 has two Ranges, both with lower and upper bounds exactly equal to *p1* (denoting that the exact same memory area is accessed for each loop iteration). Access number 3 is a memory access with two Ranges. When an array is accessed only through ranges means that the ranges define an area inside the array that is accessed completely from each iteration of the external loop of the SCoP. The area accessed by each iteration of the external loop is always the same, because the memory access pattern does not depend from the external loop iterator.

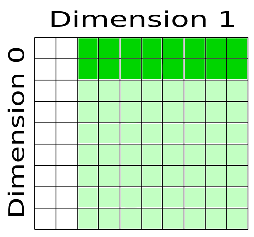


Figure 8.4: Access pattern for test application 2.

8.2.4 Jacobi method implementation

Test program number 4 shows an implementation of the Jacobi iterative matrix equation solution method [29]run for 100 iterations on a matrix (implemented through an array of double precision floating point values) 10x10 in size. The code and the results of the analysis are shown in Figure 8.6. The code clearly identifies 3 different SCoPs. Many memory accesses are performed in this algorithm.. The results in this case are even constant values (no parameters). An algorithm like this fits really well OpenMP parallelization.

8.2.5 Alternating Direction Implicit method algorithm

Test program number 5 shows an implementation of the ADI algorithm , used for solving parabolic and elliptic partial differential equations equations.[31] This is another instance of an easily parallelizable algorithm that works well with OpenMP. The program accesses two matrices of 10000 elements. Also for this algorithm the analysis brings results by exactly computing the memory access pattern for each memory accesses in the program. The program code and the result of the analysis can be seen in Figure 8.7.

```
void lu() {
  float LU[100][100];
  int i,j,k;

#pragma omp parallel for private(i,j,k) schedule(dynamic)
  for (i = 0; i < 100; ++i) {
#pragma omp single
{
        //SCoP 1
    for (j = i+1; j < 100; ++j)
      LU[i][j] = LU[i][j] / LU[i][i]; //SCoP Statement 1 (SCoP 1)
}
//SCoP 2
#pragma omp for private(j,k) schedule(dynamic)
    for (j = i+1; j < 100; ++j)
      for (k = i+1; k < 100; ++k)
        LU[j][k] = LU[j][k] - LU[i][j]*LU[i][k]; //SCoP Statement 2 (SCoP 2)

  }

}

Analysis Result:

1) Found memory access at line: 16
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: p2
2) Found memory access at line: 16
        Dimension 0 Range with LB: p5  UB: p5
        Dimension 1 Slice of size: 1
3) Found memory access at line: 16
        Dimension 0 Range with LB: p5  UB: p5
        Dimension 1 Range with LB: 0  UB: p2
4) Found memory access at line: 16
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: p2
5) Found memory access at line: 11
        Dimension 0 Range with LB: p1  UB: p1
        Dimension 1 Slice of size: 1
6) Found memory access at line: 11
        Dimension 0 Range with LB: p1  UB: p1
        Dimension 1 Range with LB: p1  UB: p1
7) Found memory access at line: 11
        Dimension 0 Range with LB: p1  UB: p1
        Dimension 1 Slice of size: 1
```

Figure 8.5: Code for Analysis test application number 3.

```
#define ITER 100
#define SIZE 10
void Jacobi() {
  double A[SIZE][SIZE];
  double B[SIZE][SIZE];
  int i,m,n;
  for (i = 0; i < ITER; ++i) {
#pragma omp parallel default(shared) shared(A,B) private(i)
    {
//SCoP 1
#pragma omp for private(m,n) schedule(dynamic)
    for (m = 0; m < SIZE; ++m)
      for (n = 0; n < SIZE; ++n)
        A[m][n] = 1.0; //SCoP Statement 1 (SCoP 1)
//SCoP 2
#pragma omp for private(m,n) schedule(dynamic)
    for (m = 0;  m < SIZE-2; ++m)
      for (n = 0; n < SIZE-2; ++n)
        //SCoP Statement 2 (SCoP 2)
        A[m][n] = B[m][n-1] + B[m][n+1] + B[m-1][n] + B[m+1][n];
//SCoP 3
#pragma omp for private(m,n) schedule(dynamic)
    for (m = 0; m < SIZE-1; ++m)
      for (n = 0; n < SIZE-1; ++n)
        B[m][n] = A[m][n]; //SCoP Statement 3 (SCoP 3)
    }
  }
}

Analysis Result:

1) Found memory access at line: 57
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 8
2) Found memory access at line: 57
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 8
3) Found memory access at line: 50
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 1  UB: 8
4) Found memory access at line: 50
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 7
... EQUAL TO ACCESS 4 ...

8) Found memory access at line: 44
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 9
```

Figure 8.6: Code for Analysis test application number 4.

```
float A[100][100];
float B[100][100];

void ADI() {

int i,j;

#pragma omp parallel
{
#pragma omp for private(i,j) schedule(dynamic)
for (i = 0; i < 100; ++i)
  for (j = 0; j < 100; ++j)
    A[i][j] = A[i][j] - B[i][j] * A[i-1][j];

#pragma omp for private(i,j) schedule(dynamic)
for (j = 0; j < 100; ++j)
  for (i = 0; i < 100; ++i)
    A[i][j] = A[i][j] - B[i][j] * A[i][j-1];

}

}

Analysis Result:

1) Found memory access at line: 18
        Dimension 0 Range with LB: 0  UB: 99
        Dimension 1 Slice of size: 1
2) Found memory access at line: 18
        Dimension 0 Range with LB: 0  UB: 99
        Dimension 1 Slice of size: 1
3) Found memory access at line: 18
        Dimension 0 Range with LB: 0  UB: 99
        Dimension 1 Slice of size: 1
4) Found memory access at line: 18
        Dimension 0 Range with LB: 0  UB: 99
        Dimension 1 Slice of size: 1
5) Found memory access at line: 13
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 99
6) Found memory access at line: 13
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 99
7) Found memory access at line: 13
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 99
8) Found memory access at line: 13
        Dimension 0 Slice of size: 1
        Dimension 1 Range with LB: 0  UB: 99
```

Figure 8.7: Code for Analysis test application number 5.

Bibliography

[1] Moore, Gordon E. (1965). *Cramming more components onto integrated circuits*

[2] Amdahl, Gene M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*

[3] OpenMP Architecture Review Board (2011). *OpenMP Application Program Interface Specification*

[4] Mark Murphy. *Loop Parallelism*

[5] LLVM Website. http://llvm.org/Users.html

[6] LLVM Website. http://llvm.org/

[7] Mohamed-Walid Benabderrahmane, Louis-Noel Pouchet, Albert Cohen and Cedric Bastoul. *The Polyhedral Model Is More Widely Applicable Than You Think*

[8] Chris Lattner (2002). *The LLVM Instruction Set and Compilation Strategy*

[9] Andrea Di Biagio, Ettore Speziale, Giovanni Agosta. *Exploiting Thread-Data Affinity In OpenMP with Data Access Patterns*

[10] Cdric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, Olivier Temam. *Putting Polyhedral Loop Transformations to Work*

[11] Tobias C. Grosser. *Enabling Polyhedral Optimizations in LLVM*

[12] Sven Verdoolaege. *Integer Set Library Manual*

[13] Louis-Nol Pouchet.
http://www.cse.ohio-state.edu/ pouchet/software/polyopt/doc/htmltexinfo/Specifics-of-Polyhedral-Programs.html

[14] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1Region.htm

[15] Polly git repository commit log.
http://repo.or.cz/w/polly-mirror.git/commit/615bd548f769f944739506fd01f653ee41ec333f

[16] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1SCEV.html

[17] The "Gang of Four". *Design Patterns:Elements of Reusable Object-Oriented Software*

[18] LLVM Assembly Language Reference. http://llvm.org/docs/LangRef.html

[19] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html

[20] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1CallGraph.html

[21] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1CallGraphNode.html

[22] Reese T. Prosser. *Applications of Boolean matrices to the analysis of flow diagrams*

[23] LLVM Documentation. http://llvm.org/docs/doxygen/html/classllvm_1_1DominatorTree.html

[24] LLVM Documentation. http://llvm.org/docs/doxygen/html/structllvm_1_1PostDominatorTree.html

[25] Donald E. Knuth. The Art of Computer Programming Volume 1: Fundamental Algorithms, third edition

[26] Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang, Tien-Hsiung Weng, Oscar Hernandez. *Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization*

[27] Chris Lattner, Dinakar Dhurjati, Gabor Greif, Joel Stanley, Reid Spencer, Owen Anderson. *LLVM Programmer's Manual*

[28] Duncan Sands. *Reimplementing llvm-gcc as a gcc plugin*

[29] Louis A. Hageman, David M. Young. *Applied iterative methods*

[30] Paht Juangphanich. *LU Factorization C code*

[31] Ryan Nong, Danny C. Sorensen. *A Parameter Free ADI-like Method for the Numerical Solution of Large Scale Lyapunov Equations*