ENABLING POLYHEDRAL OPTIMIZATIONS IN LLVM

TOBIAS CHRISTIAN GROSSER

Diploma Thesis

Programming Group
Department of Informatics and Mathematics
University of Passau

Supervisor: Prof. Christian Lengauer, Ph.D.
Second reader: Priv. Doz. Dr. Martin Griebl

Tutor: Dr. Armin Größlinger

April 2011

Tobias Christian Grosser: *Enabling polyhedral optimizations in LLVM,* Diploma Thesis, © April 2011

ABSTRACT

Sustained growth in high performance computing and the availability of advanced mobile devices increase the use of computation intensive applications. To ensure fast execution and consequently low power usage modern hardware provides multi-level caches, multiple cores, SIMD instructions or even dedicated vector accelerators. Taking advantage of those manually is difficult and often not possible in a portable way. Fortunately, advanced techniques that increase data-locality and parallelism with the help of polyhedral abstractions are known to be effective in exploiting hardware capabilities. Yet, their automatic use is currently limited. They are mostly implemented in language specific source-to-source compilers which can only optimize manually annotated code that matches a certain canonical structure. Furthermore, polyhedral optimizers often target C or CUDA code, which limits efficient communication with compiler internals and can lead to missed optimization opportunities.

In this thesis we present Polly, a project to enable polyhedral optimizations in LLVM. LLVM is an infrastructure project used in compilers for a large set of different programming languages. It is built around a low-level intermediate representation (LLVM-IR) that allows language independent optimizations. We present how Polly can apply polyhedral transformations on this representation. This includes the detection of static control parts, their translation into a $\mathbb{Z}$-polyhedra based representation, optimizations on this representation and finally, the generation of optimized LLVM-IR. We also define an interface to connect external optimizers and show a novel approach to detect parallelism which is used to generate SIMD and OpenMP code. Finally, we show in some experiments how Polly can be used to automatically apply optimizations for data locality and parallelism.

ACKNOWLEDGMENTS

I would like to thank the people who contributed to the development of Polly and continue to develop Polly with me. Hongbin Zheng worked on larger parts of the front end, the general infrastructure and the test cases, Raghesh A worked on OpenMP code generation, and Andreas Simbürger helped with the connection to CLooG.

For my academic background I want to thank Albert Cohen, Martin Griebl, Armin Größlinger, Sebastian Pop, Louis-Noël Pouchet, and Sven Verdoolaege, who largely affected this work. Those people raised my interest for polyhedral techniques, helped me with my first steps and guided me during the last years. Today they are an invaluable source of knowledge. Thank you for all the helpful discussions.

I would especially like to thank P. Sadayappan, who generously supported my work on Polly with a research scholarship at Ohio State (NSF 0811781 and 0926688), Dirk Beyer, who supported the development of the RegionInfo analysis with several university projects and Christian Lengauer, who allowed me to work on Polly for my thesis.

CONTENTS

1 INTRODUCTION 1

I BACKGROUND 5

2 LLVM 7

2.1 Architecture . . . 7

2.2 Intermediate Representation (LLVM-IR) . . . 9

2.2.1 Types . . . 9

2.2.2 Instructions . . . 11

2.3 Analysis Passes . . . 13

2.3.1 Dominator Tree . . . 14

2.3.2 Loop Information . . . 15

2.3.3 Region Information . . . 16

2.3.4 Scalar Evolution . . . 18

2.3.5 Alias Analysis . . . 20

2.4 Canonicalization . . . 21

2.4.1 Loop Canonicalization . . . 21

3 INTEGER POLYHEDRA 23

3.1 Integer Set . . . 23

3.2 Integer Map . . . 25

3.3 Properties and Operations on Sets and Maps . . . 25

II POLLY 27

4 ARCHITECTURE 29

4.1 How to Use Polly . . . 29

4.1.1 Polly's LLVM-IR Passes . . . 30

4.1.2 pollycc - A Convenient Polyhedral Compiler . . . 30

5 LLVM-IR TO POLYHEDRAL DESCRIPTION 33

5.1 What can be Translated? . . . 34

5.2 How is a SCoP Defined on LLVM-IR? . . . 35

5.3 The Polyhedral Representation of a SCoP . . . 37

5.4 How to Create the Polyhedral Representation from LLVM-IR . . . 39

5.5 How to Detect Maximal SCoPs . . . 40

5.6 Preparing Transformations . . . 43

5.6.1 LLVM canonicalization passes . . . 43

5.6.2 Create Independent Basic Blocks . . . 43

6 POLYHEDRAL OPTIMIZATIONS 47

6.1 Transformations on the Polyhedral Representation . . . 47

6.2 External Optimizers - JSCoP . . . 52

6.3 Dependency Analysis . . . 54

7 POLYHEDRAL DESCRIPTION TO LLVM-IR 57

7.1 Generation of a Generic AST . . . 57

7.2 Analyses on the Generic AST . . . 57
7.2.1 Detection of Parallel Loops . . . 59
7.2.2 The Stride of a Memory Access Relation . . . 59
7.3 Generation of LLVM-IR . . . 61
7.3.1 Sequential Code Generation . . . 62
7.3.2 OpenMP Code Generation . . . 63
7.3.3 Vector Code Generation . . . 63

III EXPERIMENTS 69
8 MATRIX MULTIPLICATION - VECTORIZED 71
9 AUTOMATIC OPTIMIZATION OF THE POLYBENCH BENCHMARKS 75
9.1 The identity transformation . . . 75
9.2 Creating Optimized Sequential Code with Pluto . . . 76
9.3 Creating Optimized Parallel Code with Pluto . . . 77
10 CONCLUSION 79

IV APPENDIX 81
List of Figures 83
List of Tables 85
List of Listings 86
List of Acronyms 87

REFERENCES 89

1

INTRODUCTION

MOTIVATION

Computation intensive applications are prevalent on high performance clusters and work stations, but gain also importance on mobile devices and web clients. There, the increased use of features like object and gesture recognition [45, 53], computational photography [32, 55] and augmented reality [52] yields to a steadily growing number of computation intensive applications. Further increases can be expected due to the integration of high quality cameras and other advanced input devices. Hence, efficient program execution is not only relevant on high performance clusters, but also on mobile devices. Efficiency is here crucial for instant user feedback and long battery life.

Modern hardware provides many possibilities to execute a program efficiently. Multiple levels of caches help to exploit data-locality and there are various ways to take advantage of parallelism. Short vector instruction sets such as Intel SSE and AVX, IBM AltiVec as well as ARM NEON can be used for fine grained parallelism. Dedicated vector accelerators or GPUs supporting general purpose computing may be used for massively parallel tasks. On platforms like IBM Cell, Intel SandyBridge and AMD Fusion similar accelerators are available tightly integrated with general purpose CPUs. Finally, an increasing number of computation cores exists even on ultra mobile platforms. The effective use of all available resources is important for high efficiency. Consequently, well optimized programs are necessary.

Traditionally, such optimizations are performed by translating performance critical parts of a software system into a low-level language like C or C++ and by optimizing them manually. This is a difficult task as most compilers support basic loop transformations as well as inner and outer loop vectorization, but as soon as complex transformations are necessary to ensure the required data-locality or to expose the various kinds of parallelism, little compiler support is available. The problem is further complicated if hints are needed by the compiler, OpenMP parallel loops need to be annotated or accelerators should be used. As a result, only domain experts are able to perform such optimizations effectively.

Even if they succeed, there remain other problems. First of all, such optimizations are extremely platform dependent and often not even portable between different microarchitectures. Consequently, programs need to be optimized for every target architecture separately. This becomes increasingly problematic, as today an application may target at the same time ARM based iPhones, Atom and AMD Fusion based netbooks as well as a large number of desktop processors, all equipped with a variety of different graphic and vector accelerators. Furthermore, manual optimizations are often even impossible. High level language features like the C++ standard template library block many optimizations. Languages such as Java, Python and JavaScript provide no support for portable low level optimizations. Even programs compiled for Google Native Client [56], a framework for portable, calculation intensive web applications, will face portability issues, if advanced hardware features are used. In brief, manual optimizations are complex, non-portable and often even impossible.

Fortunately, powerful algorithms are available to optimize computation intensive programs automatically. Wilson et al. [54] implemented automatic parallelization and data-locality optimizations based on unimodular transformations in the SUIF compiler, Feautrier [21] developed an algorithm to calculate a parallel execution order from scratch and Griebl and Lengauer [24] developed LooPo, a complete infrastructure to compare different polyhedral algorithms and concepts. Furthermore, Bondhugula et al. [13] created Pluto, an advanced data-locality optimizer that simultaneously exposes thread and SIMD level parallelism. There are also methods to offload calculations to accelerators [10, 9] and even techniques to synthesize high-performance hardware [43]. All these techniques are part of a large set of advanced optimization algorithms built on polyhedral concepts.

However, the use of these advanced algorithms is currently limited. Most of them are implemented in source to source compilers, which use language specific front ends to extract relevant code regions. This often requires the source code of these regions to be in a canonical form and to not contain any pointer arithmetic or higher level language constructs such as C++ iterators. Furthermore, the manual annotation of code that is safe to optimize is often necessary, as even tools limited to a restricted subset of C commonly ignore effects of implicit type casts, integer wrapping or aliasing. Another problem is that most implementations target C code and subsequently pass it to a compiler. The limited integration blocks effective communication between polyhedral tools and compiler internal optimizations. As a result, influencing performance related decisions of the compiler is difficult and the resulting programs often suffers from poor register allocation, missed vectorization opportunities or similar problems.

We can conclude that a large number of computation intensive programs exist, that need to be optimized automatically to be executed efficiently on current hardware. Existing compilers have difficulties with the required complex transformations, but there is a set of advanced polyhedral techniques that are proven to be effective. Yet, they miss integration in a production compiler to have significant impact.

CONTRIBUTIONS

With Polly[1] we present a polyhedral infrastructure for LLVM that supports fully automatic transformation of existing programs. Polly detects and extracts relevant code regions without any human interaction. Since Polly optimizes the LLVM intermediate representation (LLVM-IR), it is programming language independent and transparently supports constructs like C++ iterators, pointer arithmetic or goto based loops. It is built around an advanced polyhedral library [50] that supports existentially quantified variables and provides a state-of-the-art dependency analysis. Due to a simple file interface it is easily possible to apply transformations manually or to use an external optimizer.

---

[1] The word Polly is a combination of **Poly**hedral and **LL**VM. It is pronounced like Polly, the parrot.

We use this interface to integrate Pluto [13], a modern data locality optimizer and parallelizer. Thanks to integrated SIMD and OpenMP code generation, Polly automatically takes advantage of existing and newly exposed parallelism.

This thesis is organized as follows. In Chapter 2 and 3 we give some background on LLVM and $\mathbb{Z}$-polyhedra. Then we describe the concepts and techniques developed within Polly. We start in Chapter 4 with presenting the architecture of Polly and the tools developed to use it. Next, we explain in Chapter 5 the detection of relevant code regions in LLVM-IR and their translation into a polyhedral description. We also define the description itself. In Chapter 6 we present an advanced dependency analysis on this representation and we show how optimizations on it are performed. In this chapter we also define an exchange format to connect external optimizers. In Chapter 7 we describe how we regenerate LLVM-IR and how to automatically create OpenMP and SIMD parallel code. After the concepts developed within Polly are explained, we show some experiments in Chapters 8 and 9, that help to understand the impact of our current implementation. Finally, we conclude in Chapter 10.

Part I

BACKGROUND

2

LLVM

The Low Level Virtual Machine (LLVM) is a compiler framework described in Lattner and Adve [31]. It is built around the LLVM Intermediate Representation (LLVM-IR) and comes with a large variety of analysis and transformation passes. The whole framework was especially designed to optimize a program throughout its lifetime, which means at compile time, link time and even run time. Furthermore, LLVM includes back ends for static and just-in-time code generation for architectures like x86-32, x86-64, ARM or PowerPC.

LLVM based compilers exist for a variety of programming languages. There are static compilers like LLVM clang or the GCC based compilers dragonegg [1] and llvm-gcc. clang compiles C, C++ and Objective-C whereas dragonegg and llvm-gcc additionally support FORTRAN, Ada and Go. Furthermore, there are compilers for dynamic languages like Python [6], Ruby [5] and Lua [3] and for bytecode based virtual machines like Java [2] and .NET [47]. With the Glasgow Haskell Compiler (GHC) even a compiler for an all functional language includes LLVM based code generation [46]. LLVM is also used in areas like graphics shader optimization, GPGPU computing or for the AMD [33], Apple and Nvidia OpenCL compilers [34]. There exist even a notable number of projects to use LLVM for High Level Synthesis (HLS) [51, 14, 57, 40]

2.1 ARCHITECTURE

LLVM has been developed as set of libraries which implement various parts of a compiler. They can either be embedded into existing compilers to incorporate LLVM functionality or a set of LLVM command line tools can be used to access library features directly. All libraries provided by LLVM work on a common intermediate representation called LLVM-IR. It is possible to export the LLVM-IR at different stages of the compilation, to move it between tools or to modify it manually.

A good way to get an idea of existing LLVM features is to examine a classical static compilation process that uses the LLVM tools. Figure 1 shows a small program consisting of three source files that is compiled into a single executable. The source files contain code written in three different programming languages. Each source file is lowered by a language specific front end to LLVM-IR. Common frontends are clang or `llvm-gcc` for C/C++ code as well as `llvm-gfortran` for FORTRAN code. Subsequently, the `llvm-opt` tool is run on each LLVM-IR file to optimize the individual translation units and to generate an optimized LLVM-IR file. All optimizations are performed on the same intermediate representation such that a common set of optimization passes can be used for transla-

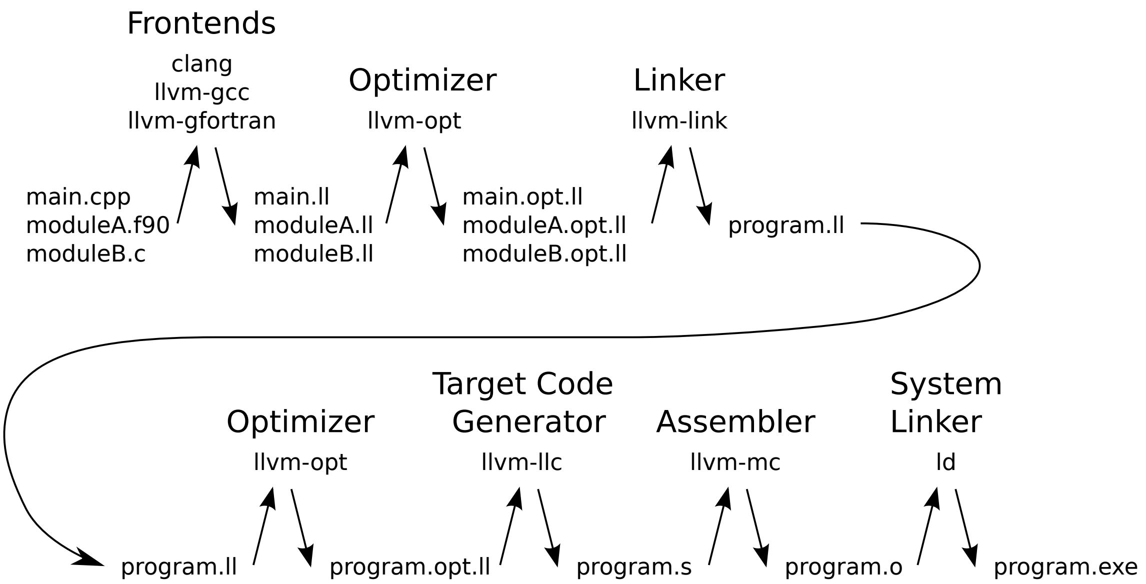

Frontends
clang
llvm-gcc
llvm-gfortran
Optimizer
llvm-opt
Linker
llvm-link
main.cpp
moduleA.f90
moduleB.c
main.ll
moduleA.ll
moduleB.ll
main.opt.ll
moduleA.opt.ll
moduleB.opt.ll
program.ll
Optimizer
llvm-opt
Target Code Generator
llvm-llc
Assembler
llvm-mc
System Linker
ld
program.ll
program.opt.ll
program.s
program.o
program.exe

Figure 1: *Static compilation using the llvm toolchain*

tion units created from different programming languages. The optimizers can be run as set of standard optimizations passes or as individually selected passes. This is especially helpful if e.g. only dead code elimination should be performed or the LLVM-IR should be canonicalized in a certain way.

After the individual parts have been optimized `llvm-link` combines them to a single module, which can be optimized again by `llvm-opt`. At this point specific link time optimizations are scheduled that for instance mark all functions as internal. In addition a set of standard optimizations is run, as they may apply again after the link time specific optimizations. As modules created from different programming languages are linked together, optimizations can cross language borders without difficulty. Therefore, it is possible to inline a function written in FORTRAN into some code written in C++.

Up to this point all transformations can be performed target agnostic. The first transformation that cannot is the lowering from LLVM-IR to target specific machine code. However, even at this stage LLVM uses a target independent code generation infrastructure that provides generic support for instruction selection or register allocation. The target code generators for the different supported architectures are based on this generic infrastructure and can be specialized by target specific optimizers. Instruction selection is in charge of selecting the best machine code instructions to implement a set of LLVM-IR instructions. Machine specific aligned or unaligned load instructions are e.g. selected to implement a generic load depending on the alignment information LLVM-IR provides. A sequence of an add and a multiply instruction can be lowered to a machine specific fused multiply add. In addition arbitrary width vector types, as provided by LLVM-IR, are lowered to machine instructions that work on vectors of target specific width.

`llvm-llc` is the tool that creates target assembler code from LLVM-IR. As LLVM code generation uses a generic machine code infrastructure, it is also possible to directly emit object files bypassing any assembler. The same infrastructure is used in the `llvm-mc` tool, which provides an assembler, a disassembler as well as ways to obtain information like the encoding of machine instructions. The machine code infrastructure can also be used by the target independent optimization passes of LLVM which obviously can take target specific information into account. Therefore, unrolling or inlining decisions can be taken based on the real instruction encoding instead of some rough estimates. This is especially important for modern CISC architectures.

Another area where the machine code infrastructure is very useful is the just in time infrastructure of LLVM. Instead of creating static object files LLVM supports just-in-time code generation such that machine code is directly emitted to memory and executed right ahead. As the LLVM just in time compiler uses LLVM-IR to describe the programs all optimization passes available for static compilation can be applied during just-in-time compilation. It is conceptually even possible to reoptimize with different parameters e.g. to especially optimize hot functions or to take advantage of knowledge obtained during the execution of the program.

Overall, LLVM provides a consistent infrastructure for the whole compilation process that is used in many important compilers. As all compilers target LLVM-IR as common intermediate representation LLVM is a great platform to write programming language and target independent optimizations.

2.2 INTERMEDIATE REPRESENTATION (LLVM-IR)

LLVM uses a common intermediate representation called LLVM-IR throughout the whole compilation process. All compilers that use LLVM as optimizer and code generator lower their input language to LLVM-IR. LLVM-IR is a typed, target agnostic, assembler like language for a register machine with an infinite amount of virtual registers. Each register can only be written once, such that register operations are in Static Single Assignment (SSA) form. As LLVM is a load/store architecture, values are transfered between memory and registers via explicit `load` or `store` operations.

There exist three representations of LLVM-IR. First there is the in-memory representation used inside the library, then there is the on-disk bytecode representation (.bc files) used e.g. for caching the output of a just-in-time compiler and finally there is the human readable assembly language representation (.ll files). All three are equivalent in terms of expressiveness. In this thesis the human readable representation is used.

This section describes the subset of LLVM-IR relevant for Polly and highlights important aspects. A full definition is available from Lattner [30].

2.2.1 *Types*

LLVM-IR is strongly typed, which means every register, function parameter, function return value or instruction has an associated type. There exist no implicit type casts. The operands provided to functions or instructions must always match the required types. The only way to change the type of a value is an explicit cast. Code without explicit casts in the LLVM-IR is always type safe. However, casts may be needed to express the type systems of more abstract languages. As a result types in LLVM-IR correspond not necessarily to the ones that exist in the language from which the LLVM-IR is generated.

The types can be divided into two major groups: primitive types and derived types. Primitive types are the basic types. Derived types are constructed by recursively composing such basic types.

There are three kinds of primitive types relevant for Polly: the integer types, the floating point types and the label type. LLVM-IR supports integer types of arbitrary but fixed bit width called `iX`, where `X` is the bit width of the integer. All optimizations and analyses are implemented on these arbitrary width integers. As not all targets have robust code generation for large bit widths, the use of types beyond `i128` is limited. In contrast to C integer types in LLVM-IR do not define signedness. The interpretation of the integer values depends on the instruction that uses them. For further information see Section 2.2.2. LLVM-IR also defines floating point types for single, double as well as quadruple precision. All available floating point types are signed. Finally, there is the label type that represents locations in the source code which can be used as a target for branch and switch instructions. Common primitive types are :

```
i1         ; Boolean value
i8         ; C (unsigned) char
i32        ; 32 bit (unsigned) integer
i64        ; 64 bit (unsigned) integer

float      ; Single precision floating point value
double     ; Double precision floating point value
x86_fp80   ; X87 80bit floating point value
fp128      ; Quadruple precision floating point value

label      ; The label of a basic block
```

There are three kinds of derived types interesting to Polly: the pointer types, the array types and the vector types. A pointer type specifies the location of an object in memory. An array type describes a set of elements arranged sequentially in memory. It has always a fixed size known at compile time. There are no variable sized multi-dimensional arrays in LLVM-IR. Variable sized arrays need to be expressed based on pointers and pointer arithmetic. A vector type represents a vector of elements. It is defined by the number of contained elements and their common primitive type. Vector types are used as operands in SIMD operations. They can have arbitrary width and will be lowered by the target back ends to the machine vector width. As this lowering is done in the target code generation, no further LLVM-IR optimizations are applied. As a result, code generation for vectors that are notably larger than the machine vector width is not optimal. Common derived types are:

```
float *              ; A pointer to a float
<4 x float> *        ; A pointer to a vector of four floats

[20 x float]         ; An array of 20 float elements
[5 x [3 x double]]   ; An array that consists of five arrays
                     ; each containing three doubles
```

```
<4 x float>          ; A vector of four floats
<16 x i8>            ; A vector of 16 C chars
```

All types mentioned in this section are also first class types. First class types can be created by LLVM-IR instructions and can be passed as function parameter or returned by a function.

2.2.2 *Instructions*

LLVM-IR has on purpose a very limited set of instructions to describe a program. It only represents common operations. Specific machine instructions are created when LLVM-IR is lowered in the back end. The set of LLVM-IR instructions needed to understand Polly is even smaller. In this section we will have a look at instructions for computations, vector management, type conversion, memory management and finally control flow instructions.

First we look at computational instructions that perform a side effect free operation on a set of operands. Operands are in this case either virtual registers or constant values. There exist three groups of computational instructions: Binary instructions, bitwise instructions and comparison instructions. Binary instructions as well as comparison instructions are defined for both floating point and integer types, whereas bitwise instructions are only defined on integer types. Furthermore, there exist corresponding vector versions of all instructions, which take integer or floating point vector types as operands and perform the same operation as the non-vector instruction, but elementwise on the whole vector. The complete set of binary instructions consists of addition, subtraction, multiplication, division and modulo operation. For integer types there exist signed and unsigned versions of the division and the modulo operation. This is not needed for floating point types, as they are always signed. Integer types are interpreted following the two's complement representation with defined overflow semantics for addition, subtraction and multiplication Hence, no special signed or unsigned versions are needed for those instructions. Various bitwise instructions allow different kinds of shifts as well as the boolean operations `and`, `or` and `xor`. Finally there are comparison instructions, which support a common set of comparisons. Some computational instructions are:

```
%result = add i32 %firstOp, %secondOp
%resultF = fmul float %firstFOp, %secondFOp
%resultOr = or i8 %firstBitSet, %secondBitSet
%vectorDiv = fdiv <8 x double> %vectorOne, %vectorTwo
%vectorCmp = icmp eq <8 x i16> %intVectorOne, %intVectorTwo
```

The instructions just described only work on virtual registers. Hence it is necessary to load values from memory into registers before the instructions are executed and to store the results of a calculation from a register back to memory. The only instructions needed for this are called `load` and `store` and move a value from a register to an address in memory or the other way around. There are two ways to obtain the address of an object in memory. Either by allocating a new one on the stack using `alloca` or by using system provided functions for memory allocation like `malloc` to allocate memory elsewhere. LLVM provides a third way called `getelementptr` which is an instruction that takes the start address of an array and returns the address of an element at a certain offset. This instruction is used for type safe memory address calculations. To calculate the offset, the type of the elements in the array needs to be provided. As LLVM does not provide types for variable sized arrays only arrays that contain fixed size elements can be modeled this way. Variable sized arrays need to be implemented by manually calculating the relevant memory offsets. Another information available for these memory instructions is the alignment of the memory accessed. If alignment information is provided, the target code generator can use more efficient aligned machine operations. Possible memory operations are:

```
%memoryAddress = alloca float, align 4
store float 3.0, float* %memoryAddress, align 4
%register = load float* %memoryAddress, align 4
%vectorPtr alloca = <4 x float>, align 32
%vectorLoad = load <4 x float> %vectorPtr, align 32
```

With the features so far, it is possible to do pure scalar calculations and pure vector calculations. However it is often needed to build a vector from a set of scalars or to extract a scalar from a vector. Therefore LLVM provides the instructions `insertelement` and `extractelement`. Furthermore, a generic shuffle instruction is provided that takes two vectors and combines their content based on a shuffle mask. These are the only vector specific instructions. All other operations on vectors are provided by generic instructions. Some vector specific operations are:

```
%resultVec1 = insertelement <4 x float> %vec, float 5.0, i32 0
%resultVec2 = insertelement <4 x float> %resultVec1, float 3.2, i32 1
%resultVec3 = insertelement <4 x float> %resultVec2, float 8.4, i32 1
%shuffleResult = shufflevector <4 x float> %resultVec2, <4 x float> %resultVec3,
                          <3 x i32> <i32 0, i32 1, i32 5>
%scalar = extractelement <4 x float> %resultVec3, i32 1

; %shuffelResult is <5.0, 3.2, 8.4>
; %scalar is 8.4
```

To allow calculations between types of different sizes or between floating point and integer types LLVM provides a set of cast and conversion instructions that provide type extensions and truncation for both integer and floating point types, with special signed and zero extension methods available for integer types. Furthermore, signed and unsigned versions for float to integer conversion are provided. Finally, a bitcast instruction is available to reinterpret a set of values without modifying the actual memory content. This can for instance be used to change a pointer to a scalar into a pointer to a vector. Common conversion instructions are:

```
%smallint = trunc i32 %largeint to %i16
%hugeint = zext i32 %largeint to %i64
%float = fptrunc double 42.0 to float

%vectorPtr = bitcast i32* %scalarPtr to <2 x i32>*
%register = load <2 x i32>* %vectorPtr
```

So far only sequentially ordered instructions have been discussed. A set of such instructions that does not include any branch is called a basic block. Each basic block is labeled with a name. To create constructs like loops or conditional control flow these basic blocks are connected by so-called terminator instructions that direct the control flow from one basic block to another. Terminator instructions are required as last instruction of a basic block and are only allowed at this place. The most common ones are `branch`, `switch` and `ret`. The first describes either an unconditional branch to a single label or a conditional branch to two labels. The `switch` is a generic branch instruction to a numbered set of labels, where the target of the branch is provided by an integer value. And finally there is the `ret` instruction to terminate the control flow in a function and to return a value. As terminator instructions are required and the targets can only be known labels, an explicit Control Flow Graph (CFG) is created. Another control flow instruction is the `phi` instruction, which merges values defined in the predecessors of a basic block into a single register value. The following code shows some examples:

```
start:
  %condition = icmp eq i32 5, 10               ; 5 == 10 is False
  br %condition, label %left, label %right     ; Branch to %right
left:
  %plusOne = add i32 0, i32 1                  ; 0 + 1 (not executed)
  br label %join                               ; Branch to %join (not executed)
right:
  %minusOne = sub i32 0, i32 1                 ; 0 - 1
  br label %join                               ; Branch to %join
merge:
  %joinedValue = phi i32 [ %plusOne, %left],   ; Copy value from %minusOne
                         [ %minusOne, %right]
  ret i32 %joinedValue                         ; return %joinedValue
```

Overall, LLVM-IR is a small language that was shown to be expressive enough to represent a wide range of programming languages and preserves enough information to apply sophisticated analysis and transformations. The small size of the language simplifies analysis and transformation on it and allows them to cover the whole language easily. Through the consistent integration of vector types it is furthermore a solid platform for machine independent vectorization.

2.3 ANALYSIS PASSES

As LLVM-IR is a relatively low-level program representation, information about higher level structures or global program state is in general not readily available. Nevertheless it is often possible to analyze LLVM-IR and to derive the information needed. This is facilitated as LLVM already provides a set of sophisticated analysis passes [4]. We will present the ones used by Polly. As an example we use the following piece of code and its CFG representation as shown in Figure 2.

```
void foo() {
  int i, a, b;
```

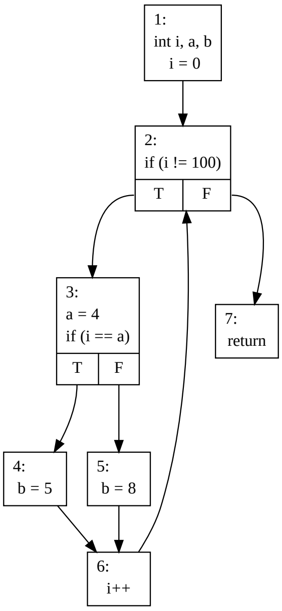


Figure 2: *CFG of the example program*

```
  for (i = 0; i != 100; i++) {
    a = 3;

    if (i == a)
      b = 5;
    else
      b = 8;
  }
}
```

2.3.1 *Dominator Tree*

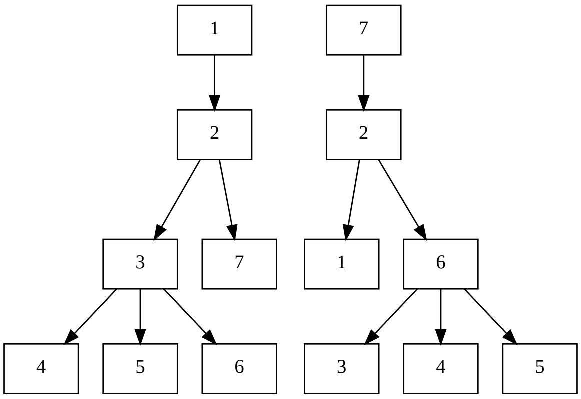
A commonly used analysis in LLVM is the dominance information[1]. It describes a relation between the different basic blocks in the CFG. Prosser [38], who developed the idea of a dominance information, introduced it with the following words:

> "We say [block] *i* dominates [block] *j* if every path (leading from input to output through the [CFG]) which passes through [block] *j* must also pass through [block] *i*."

i is called the *dominator* of *j*. It is called the *immediate dominator* of *j*, if there exists no other basic block that is dominated by *i* and that also dominates *j*. In Figure 2 basic

---

[1] Section adapted from 'The refined program structure tree' (Tobias Grosser), the description of a project developed in the class 'Softwareanalyse' 09/10 with Dirk Beyer



(a) *Dominator Tree* **(b)** *Postdominator Tree*

Figure 3: *The dominator tree for the CFG in Figure 2*

blocks 2 and 3 dominate basic block 4, but only block 3 is the immediate dominator of block 4. There is no dominance relation defined between block 4 and 5. The graph $G = (V, E)$ where V is the set of basic blocks and E is the dominance relation defined on V is called *dominator tree*. The dominator tree for the CFG in Figure 2 is shown in Figure 3. As the dominator tree is often used in LLVM, many optimizations keep it up to date. As a result new optimizations can generally expect it to be available without any further cost.

The dominance information is also defined on the reversed CFG, CFG′, which is identical to CFG but with the directions of the edges reversed. The corresponding constructs on CFG′ are called *post-dominator*, *immediate post-dominator* and *post-dominator tree*. The post-dominator tree of our example is shown in Figure 3. It shows that basic block 6 and 2 post-dominate block 5, however only block 6 is an immediate post-dominator of block 5. There is one special case that distinguishes post-dominance information from dominance information. In LLVM a sensible dominator tree is also calculated if there exist basic blocks from which there is no path to a return statement, a case occurring in the presence of non-terminating loops. Such basic blocks are not part of the post-dominator tree currently calculated by LLVM. Therefore, post-dominance information can currently not be used to optimize such programs.

2.3.2 *Loop Information*

LLVM provides an analysis to detect natural loops as described by Aho et al. [7]. Natural loops define cyclic structures in the CFG as they are created by constructs like `for`, `do..while` or `foreach`. To define a natural loop we first introduce the notion of a back edge. A *back edge* in the CFG is an edge whose source dominates the target. The *natural loop* of an edge $a \longrightarrow d$ is a subgraph of the CFG that contains d and all basic blocks that can reach a without passing through d. d is called the *loop header*. In Figure 2 edge $6 \longrightarrow 2$ is a back edge and the corresponding natural loop consists of the nodes $2, 3, 4, 5, 6$.

The natural loops of a program can be organized in a *loop tree* whose nodes are the natural loops and whose edges are defined by the subgraph relation. As a result natural

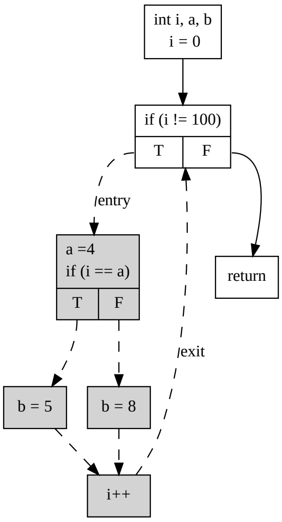


Figure 4: *A simple Region*

loop L_1 is a *child* of a natural loop L_2 , if the basic blocks in L_1 are also basic blocks in L_2.

Furthermore, LLVM detects certain special forms of natural loops. A natural loop has a *preheader* if there is a single basic block from which edges enter the loop. There exists a *preheader* of a loop, if there is a single basic block from which edges lead to the loop header.

2.3.3 *Region Information*

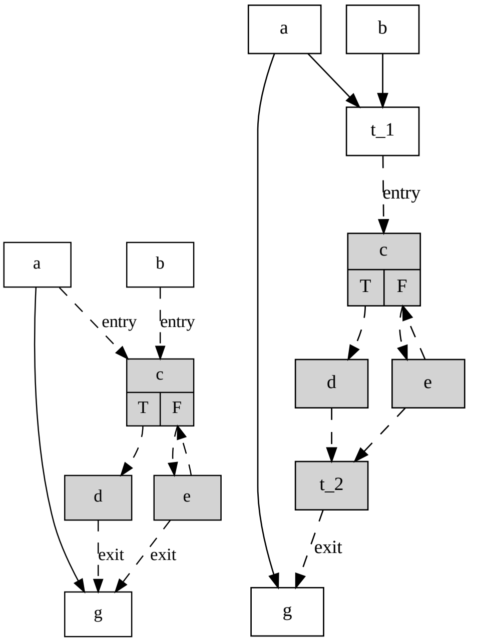
LLVM contains an analysis to detect control flow regions[2]. It was developed by us based on ideas from Johnson et al. [27], but enhanced to allow a more fine grained detection of regions. This was partially inspired by Vanhatalo et al. [49].

A *simple region* is a subgraph of the CFG that is connected to the remaining graph by only two edges, an entry edge and an exit edge. It does not influence the control flow outside of the region. Hence, it can be modeled as a function call, which can easily be replaced with a call to an optimized version of the function. A *canonical simple region* is a simple region that cannot be constructed by merging two adjacent smaller simple regions. A simple region *contains* another simple region if its nodes are a superset of the nodes of the other simple region. A tree is called a *region tree*, if its nodes are canonical simple regions and its edges are defined by the contains relation. Figure 4 highlights one of simple regions of Figure 2.

A *refined region*, or just *region*, is a subgraph of the CFG that is not necessarily a simple region, but that can be transformed into a simple region by inserting basic blocks that merge a set of edges to create a single entry or exit edge. Figure 5a shows a region that has two entry edges ($a \longrightarrow c, b \longrightarrow c$) and two exit edges ($d \longrightarrow g, e \longrightarrow g$). By introducing basic blocks t_1 and t_2 these edges are merged to create a single entry edge $t\_1 \longrightarrow c$

---

[2] Section adapted from 'The refined program structure tree' (Tobias Grosser), the description of a project developed in the class 'Softwareanalyse' 09/10 with Dirk Beyer


a
b
entry
entry
c
T
F
d
e
exit
exit
g
a
b
t_1
entry
c
T
F
d
e
t_2
exit
g

(a) *Refined region* **(b)** *After transformation*

Figure 5: *Transform a refined region to a simple region*

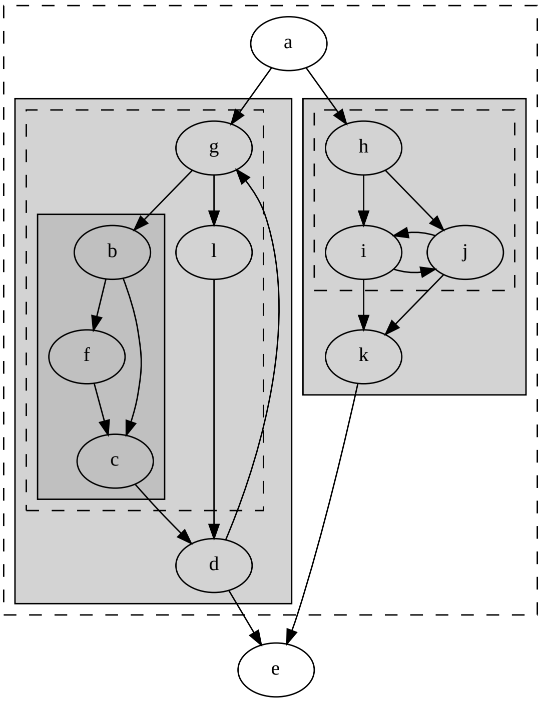

a
g
h
b
l
i
j
f
k
c
d
e

Figure 6: *Simple (solid border) and refined (dashed border) regions in a CFG.*

and a single exit edge $t\_2 \rightarrow g$. Figure 5b shows the newly created simple region. A region is *canonical* if it cannot be created by merging two adjacent regions. We also define a *contains* relation for refined regions following the definition for simple regions. Finally we define a *refined region tree*, also called *refined program structure tree*, as tree of refined regions connected by the contains relation.

Figure 6 shows a larger CFG that contains a set of different regions. Simple regions are marked with solid borders whereas refined

2.3.4 *Scalar Evolution*

The scalar evolution analysis calculates for every integer register a closed form expression describing its value during the execution of a program. This expression is called the scalar evolution of the register. It is used to abstract away the individual instructions that lead to the value of a register and to focus on the overall calculation. Scalar evolution simplifies analysis and optimizations. It is for example used to eliminate redundant instructions by comparing the scalar evolution of two registers and by eliminating one if their scalar evolutions are identical.

The analysis implemented in LLVM is based on the work of Bachmann et al. [8] on chains of recurrences and the adaption of those to compilers and induction variable analysis as described in various papers [58, 19, 37]. However, it is adapted and extended to fit the needs of the LLVM framework. One notable extension is the modeling of integer wrapping in the expressions of the scalar evolution.

2.3.4.1 *The elements of a scalar evolution*

A scalar evolution expression is constructed recursively from the elements in Figure 7. There are two base elements and a larger set of inductively defined elements. The base elements are the *integer constant* and the *unknown value*. An integer constant represents an integer known at compiler time whereas an unknown value represents an integer unknown at compile time. There are different kind of unknown values. The first is called `undef` and is used if a value is undefined in LLVM-IR and as a result will also be undefined during execution. The next is called `%parameter` and represents an integer that is unknown at compile time, but known at execution time. Parameters appear for example if a register is initialized through function parameters or by a load from memory. Finally there exist unknown values to represent target specific constants in a target independent way. These are `SizeOf`, `AlignOf` and `OffSetOf` which represent the size of types, the alignment of types and the offset of elements in a structure.

The recursively defined elements of a scalar evolution can be grouped in unary cast operations, binary operations, n-ary operations and add recurrences. There exist three unary cast operations to change the type and therefore the bit width of the value represented. The `trunk` operation reduces the bit width and `zext` or `sext` increase the bit width by applying zero or sign extension. There is one binary operation, the unsigned division $/_u$ and there are the n-ary operations addition `+`, multiplication `*`, signed maximum `smax` and unsigned maximum `umax`. All operations mentioned follow modulo integer arithmetics as defined for LLVM integer types.

- Integer constant (`42`)
- Unknown value
    - Undefined (`undef`)
    - Parameter (`%parameter`)
    - SizeOf (`sizeof(<type>)`)
    - AlignOf (`alignof(<type>)`)
    - OffsetOf (`offsetof(<structtype>, <fieldnumber>)`)
- Unary cast operations
    - Truncation (`trunc <expr>`)
    - Zero extension (`zext <expr>`)
    - Sign extension (`sext <expr>`)
- Binary operations
    - Unsigned division (`<expr>` $/_u$ `<expr>`)
- N-ary operations
    - Addition (`<expr> + <expr> + ...`)
    - Multiplication (`<expr> * <expr> * ...`)
    - Signed maximum (`<expr> smax <expr> smax ...`)
    - Unsigned maximum (`<expr> umax <expr> umax ...`)
- Add recurrence (`{<base>, +, <step>}<loop>`)

Figure 7: *The elements of a scalar evolution*

Finally, there are *add recurrences*. *Add recurrences* represent expressions that change during the evaluation of a loop. They have the format `{<base>, +, <step>}`. The *base* of an add recurrence defines its value at loop iteration zero and the *step* of an add recurrence defines the values added on every subsequent loop iteration. An add recurrence represents an affine linear expression if its step is a constant expression not containing any further add recurrences. It represents a higher order polynomial function, if the step is a non-constant expression. In general the maximal degree of a polynomial that can be expressed is limited by the nesting depth of the add recurrences.

2.3.4.2 *Code that can be analyzed*

The scalar evolution analysis in LLVM can analyze code that uses common integer arithmetic like addition, subtraction, multiplication, unsigned division. Furthermore, it supports truncation, sign extension and zero extension and is able to recognize common arithmetic idioms expressed by bitwise binary instructions or the different shift instructions. Minimum and maximum constructs expressed with conditional instructions are also commonly recovered. Loop variant expressions, which arise through the use of Φ-nodes, are detected for loops with a single entry edge and a single back edge. Finally, scalar evolution can analyze pointer arithmetic like it analyzes integer arithmetic.

Overall, scalar evolution is capable to derive closed form expressions for many of the integer variables in common programs. It is used in several important LLVM passes like redundant induction variable elimination, loop canonicalization or loop strength reduction. Hence it is as established in LLVM as more widely known analysis like the dominator information.

2.3.5 *Alias Analysis*

LLVM provides a sophisticated alias analysis infrastructure to calculate information on the relation between different memory references. It consists of several alias analysis passes that can be combined to increase the precision of the overall analysis. Each pass classifies the relation between two memory accesses as either *no alias*, *may alias*, *partial alias*, or *must alias*. *No alias* signifies that two accesses will not touch the same memory, *partial alias* means the memory ranges accessed are known to partially overlap, and *must alias* means the accessed memory is identical. In case no information about the relation between the memory accesses can be derived may alias is returned.

The different alias analysis passes available in LLVM are basic alias analysis, scalar evolution alias analysis and type based alias analysis. Basic alias analysis is the primary alias analysis implementation in LLVM and provides stateless alias analysis information. It knows for example that two different globals cannot alias. The scalar evolution alias analysis is specialized on loop structures and derives information about possible aliasing of memory references by comparing the scalar evolution expression of two loop variant memory references. Finally, there is type based alias analysis. This analysis requires additional type information from a higher level type system. As a strict high-level type system can enforce that two pointers of different types do not point to the same memory, information about possible aliasing that is not available by only analyzing, LLVM-IR can be derived.

2.4 CANONICALIZATION

There are often numerous ways to represent a calculation in a program. To be able to apply an analysis or transformation on all of them, it needs to be written in a very generic way. This yields to complex code as many special cases need to be taken into account and often even the complex code is often not generic enough to handle all representations. As a result some representations may not be optimized or may even break the analysis.

A solution to this problem is to create a simpler implementation of analysis and transformations that can only be applied on a canonical representation of the program. Code that is not in such a form is canonicalized by a set of preparing transformations.

2.4.1 *Loop Canonicalization*

Loop Simplify

The loop simplify pass transforms natural loops such that their control flow structure matches a common form. It ensures that each loop has a pre-header, which means there is a single, non-critical entry edge from outside the loop to the loop header. In addition each loop is transformed such that it has a single back edge. The loop simplify pass will also ensure that all exit edges of a loop lead to basic blocks that do not have any predecessors that are not part of the loop.

Induction Variable Canonicalization

LLVM provides a pass for induction variable canonicalization that changes every loop with an identifiable induction variable to have a have a single induction variable that counts from zero with steps of one. If additionally the number of loop iterations can be calculated, the exit condition of the loop is transformed to compare the induction variable against the number of loop iterations. If the induction variable is used outside of the loop, its use is replaced by a closed form expression that calculates the number of loop iterations.

Tail Call Elimination

A different way to express a loop is a recursive function. Even though recursive functions can be used in imperative programming languages, they are especially common for functional languages. In functional languages a special kind of recursive function, called tail recursive function, is commonly used to express loop like structures. LLVM provides a pass to transform tail call recursive functions into a imperative loop structure. As a result existing loop optimizations can be applied on programs written in functional programming languages or with functional paradigms in mind.

3

INTEGER POLYHEDRA

The main data structures used for storage and analysis of polyhedral information in Polly are integer sets and maps as implemented in the integer set library (isl) [50]. They are conceptually equivalent to $\mathbb{Z}$-polyhedra as described by Rajopadhye et al. [42], but use a different representation. In this chapter we describe the data structures and explain the functionality they provide. An explanation of how Polly represents programs with such data structures and how it uses them to perform optimizations will be given in the description of Polly itself. The definitions in this Chapter follow the ones used by Verdoolaege [50].

3.1 INTEGER SET

Definition 1 *A basic integer set is a function* $\mathcal{S} : \mathbb{Z}^n \to 2^{\mathbb{Z}^d} : s \mapsto \mathcal{S}(s)$, *where* $\mathcal{S}(s) = \{x \in \mathbb{Z}^d | \exists z \in \mathbb{Z}^e : Ax + Bs + Dz + c \geq 0\}$ *with* $A \in \mathbb{Z}^{m \times d}, B \in \mathbb{Z}^{m \times n}, D \in \mathbb{Z}^{m \times e}, c \in \mathbb{Z}^m$.

In the definition d is the number of set dimensions, n is the number of parameter dimensions and e is the number of existentially quantified dimensions. Furthermore, m defines the number of constraints in the set. A basic integer set maps a tuple of integer parameters to a set of integer tuples. It is called *universe*, if no restrictions apply e.g. $m = 0$. The elements in the basic integer set can be restricted by a finite set of affine constraints. These constraints can reference the set dimensions, the parameter dimensions and, in addition, a set of existentially quantified dimensions. Existentially quantified dimensions are only visible internally and do not change the dimensionality of a basic integer set.

An *integer set* is a finite union of basic integer sets where all elements have the same number of set and parameter dimensions. A *named integer set* is an integer set that describes a named space. Two spaces with different names reference distinct spaces, even though they may have the same dimensionality. It is not possible to apply operations on integer sets that belong to different named spaces.

An *(integer) union set* is a union of integer sets that have non-matching dimensionality or dimension names. It can conveniently be used to work with a set of related, but incomparable sets.

If the context is unambiguous, we use the term *set* to refer to an integer set, *basic set* to refer to a basic integer set and *union set* to refer to an integer union set. A (basic/union) set is called non-parametric, if it has zero parameter dimensions. We also allow some syntactic sugar in the constraints. "e mod i" (i is an integer constant) is for example

```
for (i = 1; i <= N, i++) {
  for (j = 1; j <= M && j <= 2*i, j++) {
    StmtOne(i,j)

    if (j % 2 == 0)
      StmtTwo(i,j)
  }
}
```

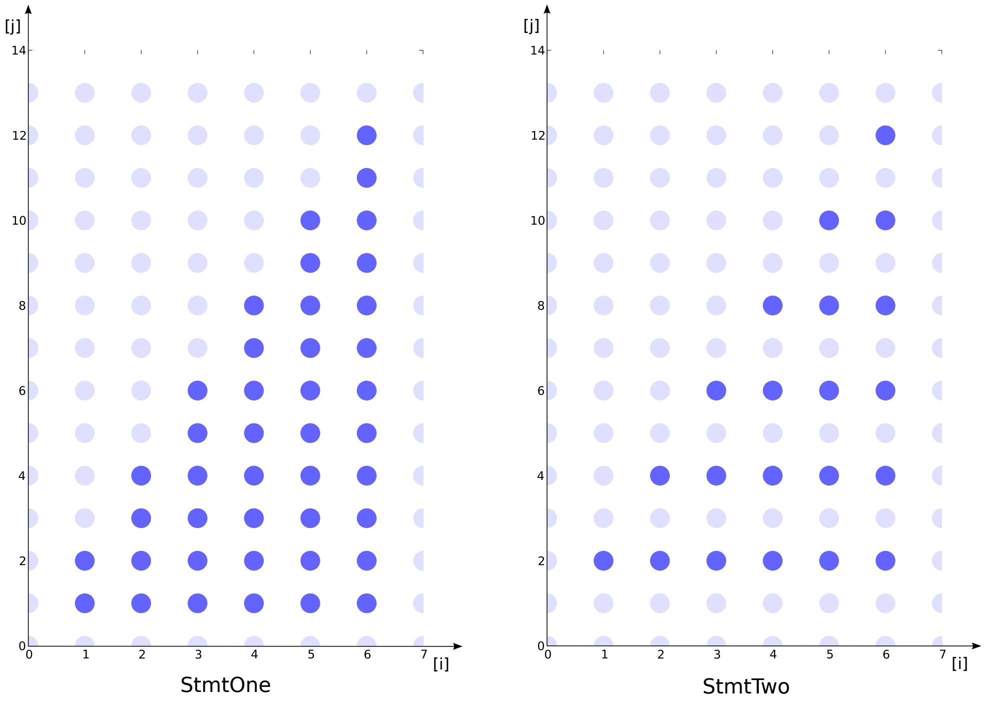


Figure 8: *A loop nest and the two basic sets used to describe the valid loop iterations.*

replaced by a set of constraints and an additional existentially quantified dimension, that enforce this modulo constraint. The same is true for *"ceild"*, *"floord"*, "$[e/i]$" (i is an integer constant). Equality constraints are also possible.

Figure 8 shows a loop nest with two statements. We represent the valid loop iterations with the two basic sets $\mathcal{S}_1 = [N, M] \rightarrow \{StmtOne[i, j] | 1 \leq i \leq N \wedge 1 \leq j \leq M \wedge j \leq 2i\}$ and $\mathcal{S}_2 = [N, M] \rightarrow \{StmtTwo[i, j] : 1 \leq i \leq N \wedge 1 \leq j \leq M \wedge j \leq 2i \wedge j \bmod 2 = 0\}$. Both are two dimensional, parametric and named. Even if they have the same number of dimensions, it is not possible to compare these sets directly. However, we can create the union of those two sets and get the union set $\mathcal{U} = [N, M] \rightarrow \{StmtOne[i, j] | 1 \leq i \leq N \wedge 1 \leq j \leq M \wedge j \leq 2i; StmtTwo[i, j] : 1 \leq i \leq N \wedge 1 \leq j \leq M \wedge j \leq 2i \wedge j \bmod 2 = 0\}$. The two basic sets shown in Figure 8 are instantiated with $N = 6$ and $M = 12$.

3.2 INTEGER MAP

Definition 2 *A basic integer map is defined as function* $\mathcal{M} : \mathbb{Z}^n \rightarrow 2^{\mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}} : s \mapsto \mathcal{M}(s)$*, where* $\mathcal{M}(s) = \{x_1 \rightarrow x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} | \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + Bs + Dz + c \geq 0\}$ *with* $A_1 \in \mathbb{Z}^{m \times d_1}, A_2 \in \mathbb{Z}^{m \times d_2}, B \in \mathbb{Z}^{m \times n}, D \in \mathbb{Z}^{m \times e}, c \in \mathbb{Z}^m$*.*

In the definition d_1 is number of input dimensions, d_2 the number of output dimensions, m the number of parameter dimensions, e is the number of existentially quantified dimensions and m the number of constraints. A *basic integer map* is a function that maps a tuple of integer parameters to a binary relation between two basic integer sets. The first set of the relation is called *domain* the second is called *range*.

An *(integer) map* is a finite union of basic maps where all elements have the same number of input, output and parameter dimensions. A *named (integer) map* is a map where either domain or range (or both) is a named space.

An *(integer) union map* is a union of integer maps that have non matching dimensionality or dimension names. It can conveniently be used to work with a map of related, but incomparable maps.

If the context is unambiguous, we use the term *map* to refer to an integer map, *basic map* to refer to a basic integer map and *union map* to refer to an integer union map. A (basic/union) map is called non-parametric, if it has zero parameter dimensions. We allow the same syntactic sugar as for integer sets.

3.3 PROPERTIES AND OPERATIONS ON SETS AND MAPS

On integer sets and maps several properties and operations are defined. We list the ones that are used within Polly.

- Properties
    - equal
    - empty
    - disjoint
    - (strict) subset
- Operations
    - complement
    - intersection
    - union
    - difference
    - lexicographic minimum/maximum
    - projection
    - inverse (maps only)
    - application
    - delta (maps only)

Most properties are well known mathematical concepts such that no further explanation is needed. In addition, they are documented in the isl user manual.[1] We detail only on the not so common operations here. The lexicographic minimum (maximum) of a map is a map that assigns to every element e in the domain the lexicographic minimal (maximal) elements in the image of e under the original map. A map can either be intersected with another map or either its domain or its range can be intersected with some set. On maps the inverse operation switches the domain and the range of a map. Furthermore, maps can be applied to the range of an integer map, the domain of an integer map or to an integer set. The difference operation calculates the elements that distinguish one set (map) from another. In contrast the delta function calculates for a map a new map that assigns to every element e in the range of the original map the differences between the elements in the image of e under the original map.

Integer sets and maps are closed under all these operations, basic integer sets and maps are not.

---

[1] `http://www.kotnet.org/~skimo/isl/manual.pdf`

Part II

POLLY

4

ARCHITECTURE

Polly is a framework that uses polyhedral techniques to optimize for data-locality and parallelism. It takes a three-step approach. First, it detects the parts of a program that will be optimized and translates them into a polyhedral representation. Then it analyses and optimizes the polyhedral representation. And finally, optimized program code is generated. To implement this structure Polly uses as a set of analysis and optimization passes that operate on LLVM-IR. We divided them into front end, middle part and back end passes.

In the front end the parts of a program that Polly will optimize are detected. Those parts, called Static Control Parts (SCoPs), are then translated into a polyhedral representation. To keep the implementation of Polly simple, Polly only detects SCoPs that match some canonical form. Any code that is not in this form is canonicalized, before it is passed to the SCoP detection.

The middle part of Polly provides an advanced dependency analysis and is the place for polyhedral optimizations. At the moment, Polly itself does not perform any optimizations, but allows the export and reimport of its polyhedral representation. The exported representation can be used to manually perform optimizations or to connect existing polyhedral optimizers to Polly. To provide an advanced polyhedral optimizer, we connected PoCC[1] with Polly. PoCC is a collection of tools that can be used to build polyhedral compilers.

In the back end of Polly the original LLVM-IR is replaced by new code that is generated following the possibly transformed polyhedral representation. At this step we calculate an imperative program structure from the polyhedral representation and create the corresponding LLVM-IR instructions. Furthermore, we detect parallel loops that can either be executed with OpenMP to take advantage of thread level parallelism or replaced with SIMD instructions, if fine grained parallelism is available. In future versions of Polly, we plan to also integrate code generation for GPU based vector accelerators. Figure 9 illustrates the overall architecture of Polly.

4.1 HOW TO USE POLLY

There exist two ways to use Polly. One is to directly run one or several of Polly's analysis and optimization passes on existing LLVM-IR files. The other is to use `pollycc`, a `gcc/clang` replacement that allows to conveniently compile C and C++ files with Polly.

---

[1] http://pocc.sf.net

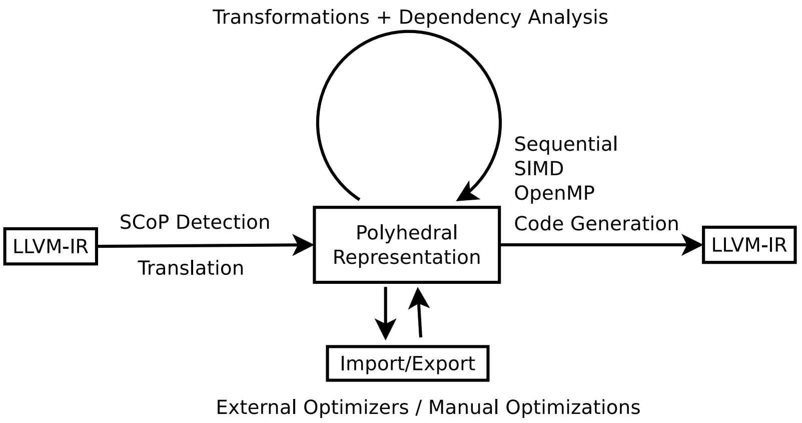

Transformations + Dependency Analysis
Sequential
SIMD
OpenMP
Code Generation
LLVM-IR
SCoP Detection
Translation
Polyhedral Representation
LLVM-IR
Import/Export
External Optimizers / Manual Optimizations

Figure 9: *Architecture of Polly*

4.1.1 *Polly's LLVM-IR Passes*

As Polly is implemented as a set of LLVM passes, it is possible to directly access each pass. LLVM provides for this the tool `opt`, which can run an arbitrary list of passes on a certain LLVM-IR file. To execute the Polly passes it is sufficient to load Polly into `opt`[2] and to select the pass that should be executed. Figure 10 gives a list of available Polly passes. Their behaviour will be explained in the subsequent parts of this thesis.

4.1.2 *pollycc - A Convenient Polyhedral Compiler*

A more convenient way to use Polly is the tool `pollycc`, which is a command line C and C++ compiler that can replace `gcc` or `clang`. It is based on `clang`, but additionally supports a set of command line options that enable Polly based polyhedral optimizations. `pollycc` automatically schedules all required canonicalization and cleanup passes, and links with run time libraries, if necessary. Hence, a user only has to select the optimizations he wants to apply and `pollycc` performs them automatically.

To understand if Polly is able to detect any SCoPs, the option `-fview-scops` shows the CFGs of a program and highlights the SCoPs found[3]. Several transformation options are available. `-fpolly` enables Polly, but does not yet enable any optimizations on the polyhedral representation. `-fpluto` uses Pluto to calculate an optimized schedule and `-ftile` additionally enables Pluto based tiling[4]. To take advantage of parallelism exposed for example by Pluto OpenMP and SIMD code generation is available through `-fparallel` and `-fvector`. In addition `-fpolly-export-jscop` and `-fpolly-import-jscop` allow to export and reimport the polyhedral representation as a JSCoP file. A complete list of available options is shown in Figure 11. Their exact behaviour will be explained in the corresponding sections of this thesis.

---

[2] Using `opt -load LLVMPolly.so ...`
[3] Requires a dot viewer available at `http://www.graphviz.org`
[4] Both require PoCC to be installed

- Front End

  polly-prepare Prepare code for Polly

  polly-region-simplify Transform refined regions into simple regions

  polly-detect Detect SCoPs in functions

  polly-analyze-ir Analyse the LLVM-IR in the detected SCoPs

  polly-independent Create independent blocks

  polly-scops Create polyhedral description of SCoPs

- Middle End

  polly-dependences Calculate the dependences in a SCoPs

  polly-interchange Perform loop interchange (work in progress)

  polly-optimize Optimize the SCoP using PoCC

  - Import/Export

    polly-export-cloog Export the CLooG input file
    (Writes a .cloog file for each SCoP)

    polly-export Export SCoPs with OpenScop library
    (Writes a .scop file for each SCoP)

    polly-import Import SCoPs with OpenScop library
    (Reads a .scop file for each SCoP)

    polly-export-scoplib Export SCoPs with ScopLib library
    (Writes a .scoplib file for each SCoP)

    polly-import-scoplib Import SCoPs with ScopLib library
    (Reads a .scoplib file for each SCoP)

    polly-export-jscop Export SCoPs as JSON
    (Writes a .jscop file for each SCoP)

    polly-import-jscop Import SCoPs from JSON
    (Reads a .jscop file for each SCoP)

  - Graphviz

    dot-scops Print SCoPs of function

    dot-scops-only Print SCoPs of function (without function bodies)

    view-scops View SCoPs of function

    view-scops-only View SCoPs of function (without function bodies)

- Back End

  polly-cloog Execute CLooG code generation

  polly-codegen Create LLVM-IR from the polyhedral information

Figure 10: *Passes available in Polly*
as printed by '`opt -load LLVMPolly.so -help`'

```
pollycc -h
usage: pollycc [-h] [-o OUTPUT] [-I INCLUDES] [-D PREPROCESSOR] [-l LIBRARIES]
               [-L LIBRARYPATH] [-O {0,1,2,3,s}] [-S] [-emit-llvm]
               [-std STANDARD] [-p] [-c] [-fpolly] [-fpluto] [-faligned]
               [-fview-scops] [-fview-scops-only] [-ftile] [-fparallel]
               [-fvector] [-fpolly-export] [-fpolly-import] [-commands] [-d]
               [-v]
               files [files ...]

pollycc is a simple replacement for compiler drivers like gcc, clang or icc.
It uses clang to compile C code and can optimize the code with Polly. It will
either produce an optimized binary or an optimized '.o' file.

positional arguments:
  files

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT             the name of the output file
  -I INCLUDES           include path to pass to clang
  -D PREPROCESSOR       preprocessor directives to pass to clang
  -l LIBRARIES          library flags to pass to the linker
  -L LIBRARYPATH        library paths to pass to the linker
  -O {0,1,2,3,s}        optimization level
  -S                    compile only; do not link or assemble
  -emit-llvm            output LLVM-IR instead of assembly if -S is set
  -std STANDARD         The C standard to use
  -p, --progress        Show the compilation progress
  -c                    compile and assemble, but do not link
  -fpolly               enable polly
  -fpluto               enable pluto
  -faligned             Assume aligned vector accesses
  -fview-scops          Show the scops with graphviz
  -fview-scops-only     Show the scops with graphviz (Only Basic Blocks)
  -ftile, -fpluto-tile  enable pluto tiling
  -fparallel            enable openmp code generation (in development)
  -fvector              enable SIMD code generation (in development)
  -fpolly-export        Export Polly jscop
  -fpolly-import        Import Polly jscop
  -commands             print command lines executed
  -d, --debug           print debugging output
  -v, --version         print version info
```

Figure 11: *pollycc command line options*

5

LLVM-IR TO POLYHEDRAL DESCRIPTION

LLVM-IR has been designed for low level optimization. However, to hide unimportant information and to focus on a high-level optimization problem a more abstract representation is often better suited. The scalar evolution analysis is for example a higher level representation used to abstract away the instructions needed to calculate a certain scalar value. It has proved to be very convenient for the optimization of scalar computations. For memory access and loop optimizations an abstraction based on a polyhedral model presented by Kelly and Pugh [29] is widely used. It hides imperative control flow structures and allows to focus on the optimization of memory access patterns. Advanced optimizers like CHiLL [15], LooPo [24] or Pluto [13] use such an abstraction to optimize for data-locality and parallelism. An extended version of this polyhedral abstraction, called the $\mathbb{Z}$-polyhedra model was presented by Gupta and Rajopadhye [26].

Polly uses a similar extended polyhedral representation, but it uses a different way to obtain it. In contrast to most optimizers, which generate a polyhedral representation by analyzing a normal programming language, Polly analyzes a low level intermediate representation. As a result, Polly is programming language independent and can transparently support many constructs that would need special attention at the programming language level. Graphite [48] takes a similar approach by analyzing Gimple, the GCC intermediate representation. Even though Graphite and Polly share the general approach, there are notable differences between both. They arise directly from the differences of the intermediate representations, but also from the fact that Polly introduced new techniques to increase the amount of code that can be optimized and to decrease the amount of dependences that can block possible transformations. Furthermore, Graphite does not yet support the $\mathbb{Z}$-polyhedra extension.

In this chapter we describe the polyhedral representation used and we explain how it is derived from LLVM-IR. This includes a definition of the program parts that can be described, an algorithm how to find such program parts and an explanation of how to create their polyhedral representation. Furthermore, we present a set of preparing transformations that we use to increase to amount of code that we detect, to remove unneeded dependences and to simplify the implementation of Polly.[1]

---

[1] The Polly front end was implemented in collaboration with Hongbin Zheng. His work was partially founded by a Google Summer of Code 2010 scholarship where he was mentored by Tobias Grosser. His work included code to verify if a region is a SCoP as well as code translating from LLVM-IR to the polyhedral representation. He was also in charge of implementing the Polly automake and cmake build system as well as parts of the test suite

```
for (int i = 0; i < 100 + n; i = i + 4) {
  A[i + 1] = B[i][3 * i] + A[i];

  for (int j = i; j < 10 * i; j++)
    A[i + 1] =  A[i] + A[i - n];
}
```

Figure 12: *A valid SCoP (validity defined on ASTs)*

5.1 WHAT CAN BE TRANSLATED?

Traditionally polyhedral optimizations work on the Static Control Parts (SCoPs) of a function. SCoPs are parts of a program in which all control flow and memory accesses are known at compile time. As a result, they can be described in detail and a precise analysis is possible. Polly currently focuses on detecting and analysing SCoPs. Extensions to support non-statically known control flow were presented by Benabderrahmane et al. [12] and can be integrated in Polly, if needed.

SCoPs are normally defined on a high-level Abstract Syntax Tree (AST) representation of a program. A common definition is the following. A part of a program is a SCoP if the only control flow structures it contains are for-loops and if-conditions. For each loop there exists a single integer induction variable that is incremented from a lower bound to an upper bound by a constant stride. Upper and lower bounds are expressions which are affine in parameters and surrounding loop induction variables, where a parameter is any integer variable that is not modified inside the SCoP. If-conditions compare the value of two affine expressions, but do not include anything else in their condition. The only valid statements are assignments that store the result of an expression to an array element. The expression itself consists of side effect free operators or function calls with induction variables, parameters or array elements as operands. Array subscripts are affine expressions in induction variables and parameters. An example of a SCoP can be seen in Figure 12.

In contrast to a pattern matching approach on high-level ASTs, Polly uses a semantic approach to detect SCoPs on LLVM-IR. In LLVM-IR high-level programming language constructs have been lowered to basic instructions. Loops are for example expressed as conditional jumps that form a control flow cycle, array accesses are expressed as pointer manipulation and affine functions are split into a set of three-address operations. To recover the necessary high-level information Polly uses existing LLVM analysis as presented in Section 2.3. A nice effect of recalculating high-level information is that it is available, even if it was not expressed explicitly in the source code of a program. As a result, Polly can not only optimize canonical programs that use specific programming language constructs, but it can also work on any code that is semantically a SCoP. Figure 13 shows two SCoPs that can be detected by Polly easily. To illustrate their semantics we provide two semantically equivalent versions that show the canonical form of the SCoPs.

SCoP One

```
i = 0;

do {
  int b = 2 * i;
  int c = b * 3 + 5 * i;

  A[c] = i;
  i += 2;
} while (i < N);
```

Semantics - SCoP One

```
for (i = 0; i == 0 || i < N; i+=2)
  A[11 * i] = i;
```

SCoP Two

```
int A[1024];
int *B = A;

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

Semantics - SCoP Two

```
int A[1024];

for (i = 0; i < 1024 ; i++)
  A[i] = 1;
```

Figure 13: *Two valid SCoPs (validity defined based on semantics) and their canonical counterparts*

5.2 HOW IS A SCOP DEFINED ON LLVM-IR?

In Polly we define a SCoP in LLVM-IR such that we can reuse the concepts developed for high-level AST based SCoPs. Hence we detect parts of a program that are semantically equivalent to a high-level language SCoP. A SCoP in LLVM-IR is a subgraph of the CFG, which forms a simple region as defined in Section 2.3.3. In addition, it contains only well structured control flow, has no unknown side effects and can generally be represented in the polyhedral model. To ensure this, we verify the control flow in the region and we check that the region does not contain any instructions that we cannot represent. For most instructions this is simple, however two topics introduce some difficulties. The possible aliasing of memory references and the need to derive affine access functions.

The control flow in a SCoP is a nest of structured conditional branches and natural loops. Any other control flow instructions, irreducible control flow or infinite loops are not allowed. We verify this on LLVM-IR by disallowing switch instructions, indirect branch instructions and function returns. The only allowed control flow instructions are conditional and unconditional branch instructions. To ensure that conditional branch instructions do not form any unstructured code, they are only allowed in two places. At the exit of a natural loop and at the head of a structured conditional branch. Natural loops are detected by using the loop information as introduced in Section 2.3.2. Polly allows only loops with a single exit edge, as other loops are difficult to model. Furthermore, structured conditional branches are enforced by verifying that each branch not exiting a loop is always the entry block of a refined region. This ensures that there are no jumps into a conditional branch that bypass its header.

To model the control flow in a SCoP statically it is necessary to know the number of times each loop is executed and to described this number with an affine expression. The natural loop analysis together with the scalar evolution analysis introduced in Section 2.3.4 can describe the number of loop iterations as a scalar evolution expression. In case a certain loop cannot be analyzed and consequently its loop iteration count is unknown, the loop can not be part of a SCoP. If the loop iteration count is available, we need to check, if the scalar evolution expression describing it can be translated into an affine expression. This is the case if it only consists of integer constants, parameters, additions, multiplications with constants and add recurrences that have constant steps. If parameters appear in the expression, we need to check that they are defined outside of the SCoP and consequently do not change during the execution of the SCoP. We currently do not allow minimum, maximum, zero extend, sign extend or truncate expressions even though they can conceptually be represented in the polyhedral model. A conditional branch is valid in a SCoP, if its condition is an integer comparison between two affine expressions. Again, we use the scalar evolution analysis to get the scalar evolution expressions for each operand of the comparison and check whether each can be represented as an affine expression.

To decide whether an instruction is valid in a SCoP, we verify that either no unknown side effects may happen or all side effects are known and can be represented in the polyhedral model. Computational instructions, vector management instructions and type conversion instructions in LLVM-IR are side effect free and can consequently appear in a valid SCoP. Function calls are allowed if LLVM provides the information they are side effect free and always return. The same holds for compiler internal intrinsics. Exception handling instructions on the other hand are always invalid, as they describe control flow that we cannot model statically. Load and store instructions are the only instructions that can access memory. We currently require that a memory access can be described by an expression that is affine in the number of loop iterations and parameters. We check this criterion by analyzing the scalar evolution of the pointer that defines the memory location accessed. It must be affine following the conditions previously described. Additionally, the scalar evolution expression must contain a single parameter. This parameter is used as the base address of the memory access and defines the array that is accessed.

Base addresses in a SCoP must reference distinct memory spaces or they must be identical. A memory space in this context is the set of memory elements that can be accessed by adding an arbitrary offset to the base address. In case this is ensured, we can model the memory accesses as accesses to normal, non intersecting arrays. Fortunately, LLVM provides a set of alias analyses as described in Section 2.3.5, which give us exactly this information. In case two base addresses are analyzed as *must alias*, they are identical and, in case they are analyzed as *no alias*, the memory spaces that can be reached from them do not intersect. This may sound incorrect, as obviously a normal program has just one large address space and consequently a base address plus an integer offset is enough to reach any element in this space. However, programming languages like C provide sufficient information to ensure in many cases that an address derived from one base address cannot yield an address derived from another base address.

The remaining restrictions are not conceptually necessary, but they simplify the overall implementation of Polly. We require that every loop has a single, canonical induction variable that starts at zero and is incremented with unit stride. Furthermore, any scalar variable referenced must either be defined in the basic block it is used, it must be a loop induction variable or it must be defined outside the SCoP. This ensures that no scalar dependences exist between two different basic blocks. We later describe a preprocessing pass that can remove scalars that are used across basic blocks.

5.3 THE POLYHEDRAL REPRESENTATION OF A SCOP

Polly uses an abstract polyhedral description to represent a SCoP. It is based on integer sets and maps as described in Section 3.1.

A polyhedral *SCoP* $S = (context, [statements])$ is a tuple consisting of a context and a list of statements. The context is an integer set that describes constraints on the parameters of the SCoP like the fact that they are always positive or that there are certain relations between parameters.

A *statement* $Stmt = (name, domain, schedule, [accesses])$ is described in Polly by a quadruple consisting of a name, a domain, a schedule and a list of accesses. It represents a basic block BB_{Stmt} in the SCoP and is the smallest unit that can be scheduled independently.

The *domain* of a statement is a named integer set that describes the set of different loop iterations in which a statement is executed. Its name corresponds to the name of the statement. It has as many parameter dimensions as there are parameters in the SCoP. The number of set dimensions is equal to the number of loops that contain BB_{Stmt} and that are contained in the SCoP. Each set dimension describes one loop induction variable, where the first dimension describes the outermost loop. An iteration vector is a single element of the domain. Together with a statement it defines a *statement instance*.

The *schedule* of a statement is an integer map that assigns to each iteration vector a multi-dimensional point in time. In the final code this time defines the execution order of different statement instances. A statement instance (S_1, v_1) is executed before a statement instance (S_2, v_2), if the iteration vector v_1 is lexicographic smaller than the iteration vector v_2. It is valid to assign to an iteration vector no execution time. In this case the corresponding statement instance is not executed at all. Assigning to one iteration vector multiple execution times is at the moment not supported.

An access $A = (kind, relation)$ is a pair that consists of the kind of the access and the access relation. The kind is either *read*, *write* or *may write*. The access relation is an integer map that maps from the domain of a statement into a named, possibly multi-dimensional memory space. The name of the memory space is used to distinguish between accesses to distinct memory spaces. The access relation can be an affine function, but also any other relation that can be expressed with integer maps. Hence, an access may touch either a single element or a set of memory elements. If two accesses point to memory spaces with different names.

Figure 14 shows an example of a high-level program represented as a SCoP. It shows the domains, schedules and the accesses for each of the statements. For $Stmt_{A_1}$ the domain is one-dimensional as the statement is surrounded by only one loop. For this loop the constraints $i \geq 0$ and $i \leq N$ are added to the iteration space. In addition the constraint $i \leq N - 50$ is added as $Stmt_{A_1}$ is also part of a conditional branch. The some holds for $Stmt_{A_2}$, but as $Stmt_{A_2}$ is part of the else-branch of the condition the reverted

```
void scop(long N) {
  long i, j;

  for (i = 0; i < N; i++) {
    if (i <= N - 50)
      A[5*i] = 1;                // Stmt_A1
    else
      A[3*i] = 2;                // Stmt_A2

    for (j = 0; j < N; j++) {
      B[i][2*j] = 3;                // Stmt_B
    }
  }
}
```

$$Context = [N] \rightarrow \{\}$$

$$\begin{aligned}
\mathcal{D}_{Stmt_{A_1}} &= [N] \rightarrow \{Stmt_{A_1}[i] : i \geq 0 \wedge i \leq N \wedge i \leq N - 50\} \\
\mathcal{S}_{Stmt_{A_1}} &= \{Stmt_{A_1}[i] \rightarrow \Theta[0, i, 0, 0, 0]\} \\
\mathcal{A}_{Stmt_{A_1}} &= \{Stmt_{A_1}[i] \rightarrow A[5i]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}_{Stmt_{A_2}} &= [N] \rightarrow \{Stmt_{A_2}[i] : i \geq 0 \wedge i \leq N \wedge i > N - 50\} \\
\mathcal{S}_{Stmt_{A_2}} &= \{Stmt_{A_2}[i] \rightarrow \Theta[0, i, 1, 0, 0]\} \\
\mathcal{A}_{Stmt_{A_2}} &= \{Stmt_{A_2}[i] \rightarrow A[3i]\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}_{Stmt_B} &= [N] \rightarrow \{Stmt_B[i] : i \geq 0 \wedge i \leq N \wedge j \geq 0 \wedge j \leq N\} \\
\mathcal{S}_{Stmt_B} &= \{Stmt_B[i, j] \rightarrow \Theta[0, i, 2, j, 0]\} \\
\mathcal{A}_{Stmt_B} &= \{Stmt_B[i, j] \rightarrow B[i][2j]\}
\end{aligned}$$

Figure 14: *An example of a SCoP and its polyhedral representation.*

constraint $i > N - 50$ is added to its domain. $Stmt_B$ is surrounded by two loops, so a two-dimensional domain is created that contains constraints for both loops. The schedules of the statements map each statement iteration into a five-dimensional timescale, which ensures the same execution order as in the original code. Furthermore, with $\mathcal{A}_{Stmt_{A_1}}$, $\mathcal{A}_{Stmt_{A_2}}$ and $\mathcal{A}_{Stmt_B}$ three memory accesses are defined. The first two represent accesses to a one-dimensional array A, the last one an access to a two-dimensional array B.

5.4 HOW TO CREATE THE POLYHEDRAL REPRESENTATION FROM LLVM-IR

A region of LLVM-IR code that forms a valid SCoP is translated into a polyhedral representation by treating it conceptually as a high-level SCoP. We initialize the context of the SCoP with an unconstrained integer set. This integer set has zero set dimensions and as many parameter dimensions as there are parameters in the SCoP. To determine the number of parameters the scalar evolution of all addresses used in load or store instructions is calculated as well as a scalar evolution expression describing the iteration count for every loop and one for each value used in a condition of a branch instruction. The scalar evolution expressions are then scanned for the largest subexpressions, that are not modified inside the region. These form the parameters of the SCoP. Polly does not yet calculate relations between parameters, such that the context remains currently unconstrained.

The statements of a SCoP are formed by the different basic blocks in the CFG. As, besides memory accesses, basic blocks are side effect free, we only create polyhedral statements for the basic blocks that contain at least one load or store instruction. For a basic block BB_{Stmt} the *name* of a statement is a unique string set to the name of BB_{Stmt}.

The domain of a statement for BB_{Stmt} is created as follows. We add for each loop surrounding BB_{Stmt} one constraint $iv_i \geq 0$ as its lower bound and another constraint $iv_i \leq "number\ of\ loop\ iterations" - 1$ as its upper bound. Furthermore, if BB_{Stmt} is in a conditional branch, we add the corresponding condition to the domain of the statement. This is the case, if there exists a basic block BB_{Cond} that has a conditional statement, dominates BB_{Stmt} and the immediate postdominator of BB_{Cond} post dominates BB_{Stmt}, but is not BB_{Stmt}. The same property is used to understand, if a basic block that is part of a loop is before or after the basic block exiting the loop. In case it is after the exiting block, the exiting condition is also added to the domain to model the fact that BB_{Stmt} is not executed during the last loop iteration.

The initial schedule for each statement has as many input dimensions I_i as the domain has set dimension and it has $2d + 1$ output dimensions O_o, where d is the maximal loop depth in the SCoP. It contains a set of constraints $I_i = O_{2i+1}$ that set the values of the odd output dimensions to the values of the input dimensions. These constraints enforce for different instances of a single statement the same execution order as in the original loop nest. To ensure the correct execution order between instances of different statements the even dimensions are used to define the textual order of the statements. The value at dimension $2l$ is determined by topologically sorting the elements surrounded by exactly l loops. These can either be basic blocks or control flow regions that represent loops at level $l + 1$. For all basic blocks contained in loop regions at level $l + 1$ the value of dimension $2l$ is set to the value assigned to the loop region. In case no loop exists to define a dimension, the dimension is set to zero.

To calculate the accesses of a statement Polly analyzes all load and store instructions. Load instructions are represented as read accesses and store instructions as write accesses. For each load and store instruction we extract the scalar that defines the memory address and retrieve its scalar evolution. The scalar evolution expression is then translated into an affine expression. Here we use the parameter in the scalar evolution expression as the base address of the memory access. This base address is also used to name the memory space which ensures that accesses to different parts of the memory are modeled as accesses to different arrays. Polly currently only creates single-dimensional access functions. This is sufficient to represent single-dimensional arrays, but also multi-dimensional accesses, with statically know size. Support for multi-dimensional variable-size arrays is planned.

Figure 15 shows the steps from C code over LLVM-IR to our polyhedral representation. It starts with C code that is then lowered to a set of basic blocks. These blocks contain LLVM-IR instructions that calculate loop bounds, induction variables and array access functions. For instance the address of the memory accessed in $Stmt_B$ is calculated and the result of this calculation in stored in the virtual register $\%scevgep$. To derive the access function from $\%scevgep$ we calculate the scalar evolution expression $\{@B, +, (2 * sizeof(float))\}$`<%bb1>`, which describes the memory address accessed. This expression is then split into a base element B and a single dimensional access function $2 \times i$, where i is a virtual induction variable counting the number of loop iterations. As $\%indvar$ is a canonical induction variable, its value is equivalent to i. To calculate the domain of the statement we call the scalar evolution analysis and ask for the number of times the loop back edge is executed. In this example the resulting expressing is the integer constant 99. To derive the number of loop iterations we simply add one. With the knowledge, that canonical induction variables always start at zero, we get the domain $\{Stmt_B[i] : 0 \leq i < 100\}$.

5.5 HOW TO DETECT MAXIMAL SCOPS

Polly uses a structured approach to detect a set of maximal SCoPs. In general a function may contain various SCoPs which overlap or are nested. Optimizing overlapping or nested SCoPs is difficult. A SCoP that was already optimized by Polly, can sometimes not be recognized again or, in case it can, code that was already optimized will be optimized again. To avoid such redundant calculations and to maximize the space of possible transformations inside a SCoP, we detect SCoPs that are maximal and that cannot be enlarged further.

Polly uses the region tree described in Section 2.3.3 to detect maximal SCoPs. The first step is to find a set of maximal canonical regions that are SCoPs. This is done by walking down the region tree. Starting from the largest possible canonical region we descend into the tree. If a region forms a SCoP, we add it to the set of valid SCoPs and do not further descend in this part of the region tree. In case a region is no valid SCoP, we continue the search by examining its children. After the walk on the region tree, we have a set of maximal, canonical regions that form valid SCoPs. Yet, a SCoP may also be described by a non-canonical region. To get the set of maximal, not necessarily

```
void scop() {
  for (i = 0; i < 100; i++)
    B[2*i] = 3;               // Stmt_B
}
```

```
define void @scop() {
bb:
  br label %bb1

bb1: ; Basic block of Stmt_B
  %indvar = phi i64 [ 0, %bb ], [ %indvar.next, %bb1 ]
; %indvar -> {0,+,1}<%bb1>
  %tmp = mul i64 %indvar, 2
  %scevgep = getelementptr [100 x float]* @B, i64 0, i64 %tmp
; %scevgep -> {@B,+,(2 * sizeof(float))}<%bb1>
  store float 3.000000e+00, float* %scevgep, align 8
  %indvar.next = add i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, 100
  br i1 %exitcond, label %bb2, label %bb1

bb2:
  ret void
}

; Determining loop execution counts for: @scop
; Loop %bb1: backedge-taken count is 99
; Loop %bb1: max backedge-taken count is 99
```

$$
\begin{aligned}
\mathcal{D}_{Stmt_B} &= [N] \rightarrow \{Stmt_B[i] : 0 \wedge i < 100\} \\
\mathcal{S}_{Stmt_B} &= \{Stmt_B[i] \rightarrow \Theta[0, i, 0]\} \\
\mathcal{A}_{Stmt_B} &= \{Stmt_B[i] \rightarrow B[2i]\}
\end{aligned}
$$

Figure 15: *Translation from C over LLVM-IR to polyhedral shown on a simple example*

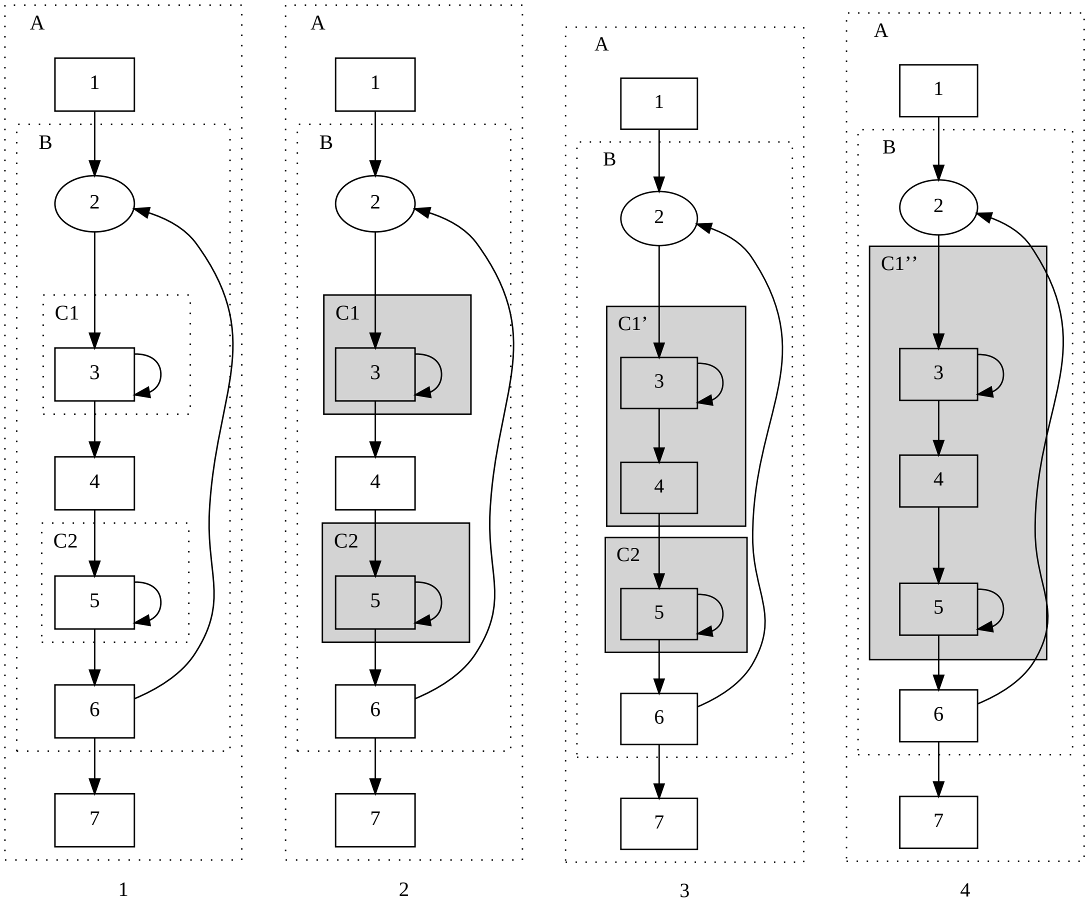


Figure 16: *Region based SCoP detection*

canonical regions that form SCoPs we enlarge the canonical regions by merging them with subsequent regions. This is done as long as the resulting regions still form valid SCoPs. When no region can be further extended, we have a set of maximal SCoPs.

Figure 16 shows in steps how we detect the set of maximal SCoPs in a function. We start at the outermost region A. It does not form a valid SCoP, as it contains basic block two, which is marked as difficult. We than check its child region, region B. It is also invalid, as it still contains basic block two. However, when we try the children of B, $C1$ and $C2$, we successfully find two regions that are valid SCoPs. These are the largest canonical regions that form valid SCoPs. To further increase their size we start to merge regions. We begin with $C1$ and merge basic block four to create $C1'$. This is valid as basic block four forms a (very simple) region. Next, we merge region $C2$. The resulting $C1''$ now contains basic block three, four and five. It cannot be enlarged further, as there exists no region that starts at basic block 6. Even basic block 6 forms no region, as it has two exit edges, that terminate at different basic blocks. Hence, for our example the set of maximal SCoPs is $\{C1''\}$.

5.6 PREPARING TRANSFORMATIONS

A set of preparing transformations is necessary to increase the coverage of Polly. Polly's front end is on purpose kept simple and often handles LLVM-IR that is in a canonical form. To ensure that LLVM-IR which does not match this form can be optimized, the LLVM-IR of a program is canonicalized before it is passed to the SCoP detection. Canonicalization is done by a set of transformations already available in LLVM and some transformations especially developed for Polly.

5.6.1 *LLVM canonicalization passes*

The set of LLVM transformation and canonicalization passes used in Polly is derived from the first half of the passes used in `clang -O3`. It contains a basic alias analysis, memory to register promotion, simplification of library calls, instruction simplification, tail call elimination, loop simplification, loop closed ssa form calculation, loop rotation, loop unswitch, and induction variable canonicalization. The most important passes have been described in Section 2.4. Furthermore, a pass to transform refined regions into simple regions is used. This pass is conceptually not necessary, but it simplifies the implementation of Polly.

5.6.2 *Create Independent Basic Blocks*

The independent block pass in Polly removes unnecessary inter basic block dependences introduced by scalar values and creates basic blocks that can be freely scheduled. Furthermore, the independent block pass promotes scalar dependences that cannot be removed into loads and stores to single element arrays. This ensures that the only scalar dependences that remain are references to induction variables or parameters. As induction variables and parameters are replaced during code generation, Polly does not need to perform any special handling for scalar dependences. All basic blocks can be moved freely during code generation.

Unnecessary dependences are regularly introduced due to calculations of array indexes, branch conditions or loop bounds which are often spread over several basic blocks. To remove such dependences, we duplicate all trivial scalar operations in each basic block. This means, if an operation does not read or write memory and does not have any side effects, its results are not transfered from another basic block, but it is entirely recalculated in each basic block. The only exceptions are PHI-Nodes, parameters and induction variables, which are not touched at all.

The recalculation of scalar values introduces a notable amount of redundant code and would normally slow down the code a lot. However, as will be later shown in Section 9, scheduling the normal LLVM cleanup passes after Polly is sufficient to remove these redundant calculations.

Figure 17 shows a SCoP that has several scalar dependences from bb1 to bb2, bb3, bb4 and bb5 which block possible transformations. They are introduced through the scalar variables $\%scevgepA$, $\%scevgepB$, $\%i.0$ and $\%valAdd$. By applying the independent block pass we move trivial calculations directly into bb2, bb3 and bb4. As can be seen in Figure 18, this introduces redundant calculations for $\%scevgepA.1$ and $\%scevgepA.2$.

However, the updated basic blocks now only depend on the value of the induction variable. The scalar dependency introduced by %*valAdd* cannot be removed by moving its calculation to bb5, as the load from %*scevgepB* cannot be moved. Consequently this scalar dependency cannot be eliminated, but it is promoted to a memory dependency. Overall the independent block pass removes all scalar dependences and it ensures that only the dependences that cannot be eliminated will show up in the final memory dependency graph.

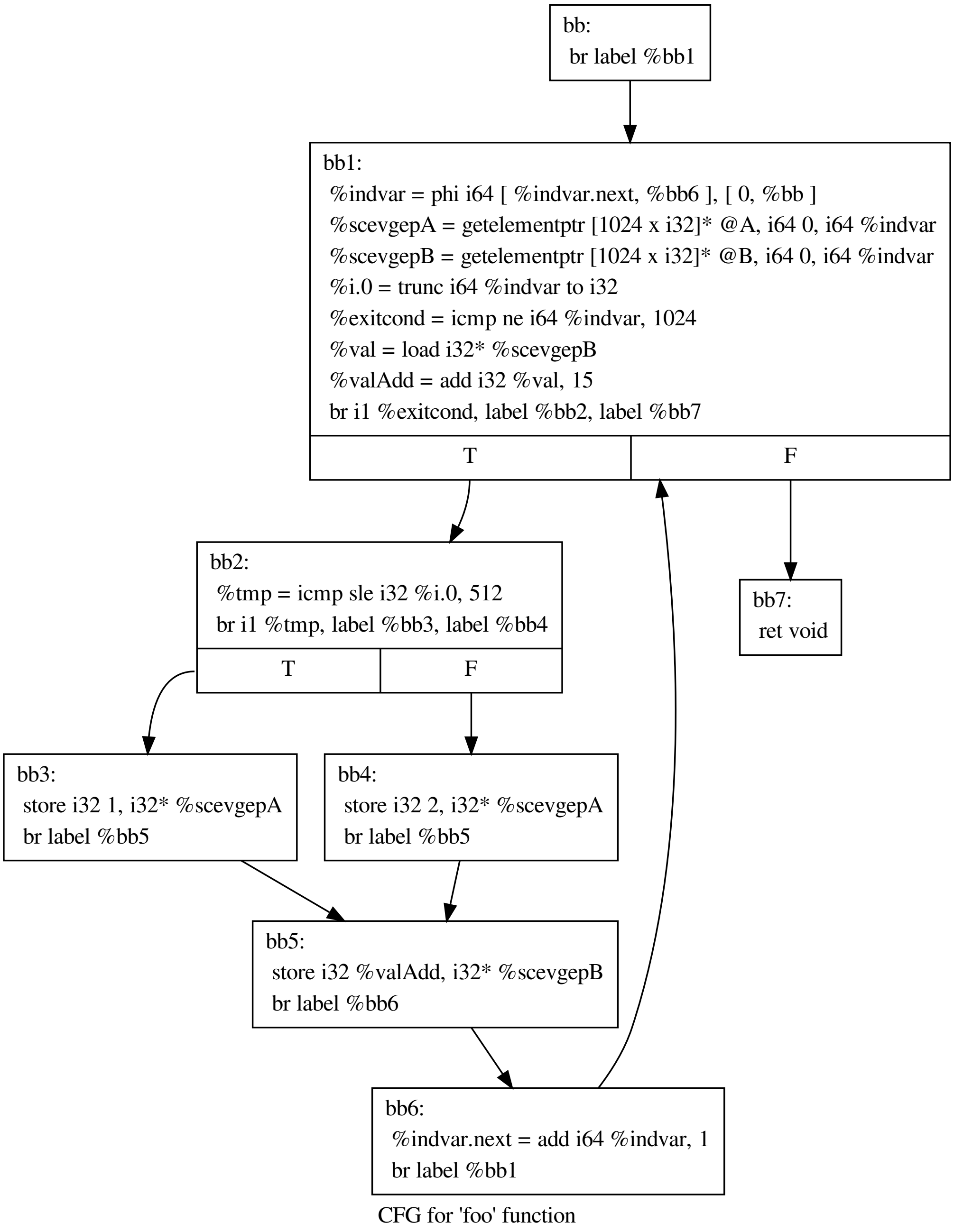
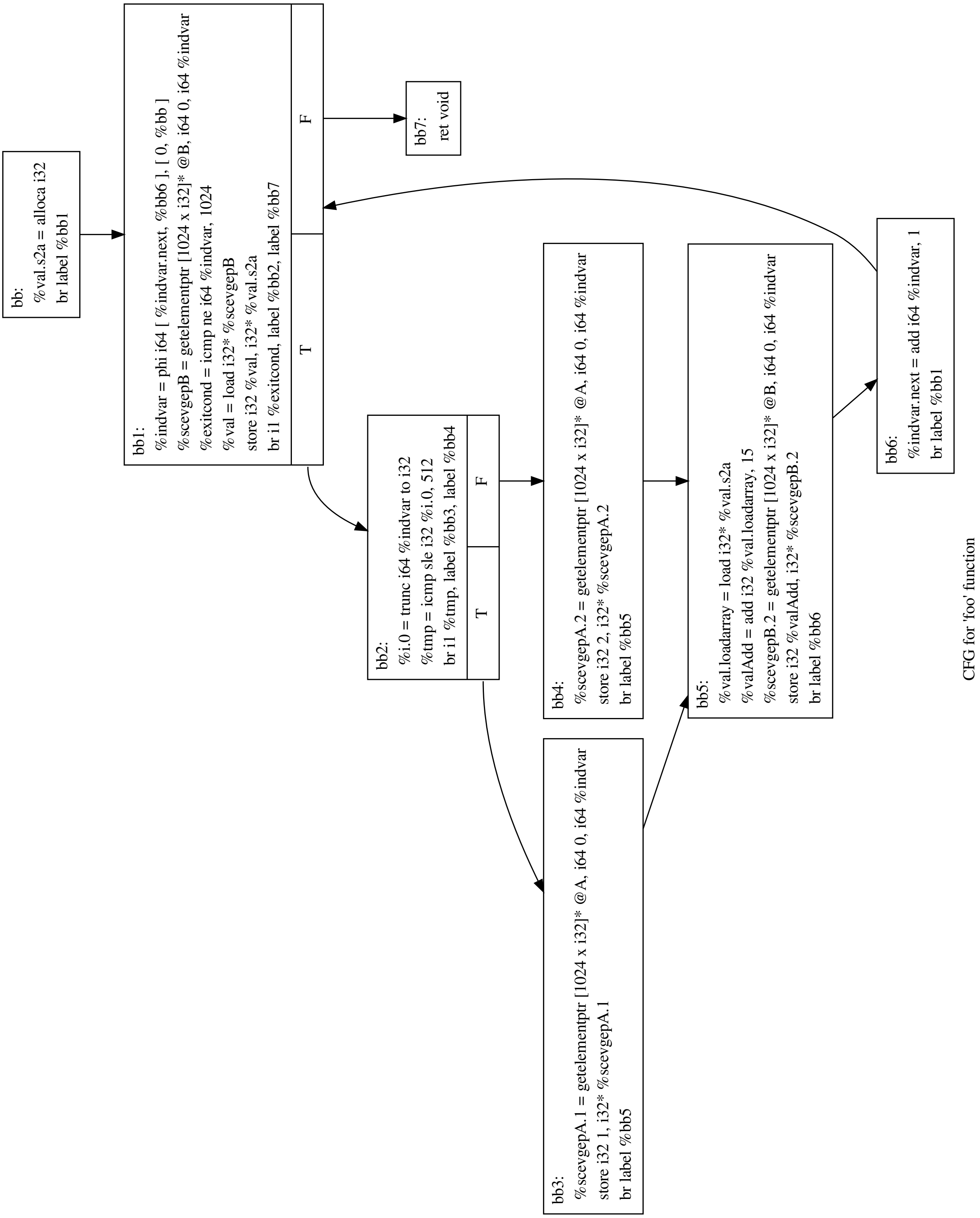

bb:
br label %bb1
bb1:
%indvar = phi i64 [%indvar.next, %bb6], [0, %bb]
%scevgepA = getelementptr [1024 x i32]* @A, i64 0, i64 %indvar
%scevgepB = getelementptr [1024 x i32]* @B, i64 0, i64 %indvar
%i.0 = trunc i64 %indvar to i32
%exitcond = icmp ne i64 %indvar, 1024
%val = load i32* %scevgepB
%valAdd = add i32 %val, 15
br i1 %exitcond, label %bb2, label %bb7
T
F
bb2:
%tmp = icmp sle i32 %i.0, 512
br i1 %tmp, label %bb3, label %bb4
T
F
bb7:
ret void
bb3:
store i32 1, i32* %scevgepA
br label %bb5
bb4:
store i32 2, i32* %scevgepA
br label %bb5
bb5:
store i32 %valAdd, i32* %scevgepB
br label %bb6
bb6:
%indvar.next = add i64 %indvar, 1
br label %bb1
CFG for 'foo' function

Figure 17: *Simple SCoP before independent block pass*



```
bb:
 %val.s2a = alloca i32
 br label %bb1

bb1:
 %indvar = phi i64 [ %indvar.next, %bb6 ], [ 0, %bb ]
 %scevgepB = getelementptr [1024 x i32]* @B, i64 0, i64 %indvar
 %exitcond = icmp ne i64 %indvar, 1024
 %val = load i32* %scevgepB
 store i32 %val, i32* %val.s2a
 br i1 %exitcond, label %bb2, label %bb7

bb7:
 ret void

bb2:
 %i.0 = trunc i64 %indvar to i32
 %tmp = icmp sle i32 %i.0, 512
 br i1 %tmp, label %bb3, label %bb4

bb4:
 %scevgepA.2 = getelementptr [1024 x i32]* @A, i64 0, i64 %indvar
 store i32 2, i32* %scevgepA.2
 br label %bb5

bb3:
 %scevgepA.1 = getelementptr [1024 x i32]* @A, i64 0, i64 %indvar
 store i32 1, i32* %scevgepA.1
 br label %bb5

bb5:
 %val.loadarray = load i32* %val.s2a
 %valAdd = add i32 %val.loadarray, 15
 %scevgepB.2 = getelementptr [1024 x i32]* @B, i64 0, i64 %indvar
 store i32 %valAdd, i32* %scevgepB.2
 br label %bb6

bb6:
 %indvar.next = add i64 %indvar, 1
 br label %bb1
```

Figure 18: *Simple SCoP after independent block pass*

6

POLYHEDRAL OPTIMIZATIONS

After showing how SCoPs are represented in Polly, we describe how they can be optimized. Basically there are two ways to optimize a SCoP. One is changing the execution order of the statement instances in the SCoP, the other is changing the memory locations they access. Polly currently focuses on changes of the execution order. Classical loop transformations like interchange, tiling, fusion and fission but also advanced transformations [13] change the execution order. Support for optimizations that change data accesses as proposed by Kandemir et al. [28] and Clauss and Meister [16] is planned.

In Polly changes to the execution order are expressed by modifying the schedules of the statements. Access relations and iteration domains are read-only. This seems to be obvious, as only the schedule is defining the execution times. However, previous approaches as for example presented by Girbal et al. [23] had difficulties to express transformations like tiling or index set splitting without changes to the domain. Those difficulties do not arise in Polly, as the integer maps used to define the schedules are expressive enough to describe those transformations.

There are two ways the schedules can be changed. Either they can be replaced by schedules that are recalculated from scratch or the schedules can be modified by a set of transformations. In Polly a transformation is an integer map that maps from the original execution time to a new execution time. It is performed by applying it to the schedules of the statements that should be transformed. Two transformations can be composed by applying the second on the range of the first.

In this Chapter we describe how classical loop transformations can be expressed, how optimizations can be applied manually, how external optimizers can be connected and how dependency analysis is performed.

6.1 TRANSFORMATIONS ON THE POLYHEDRAL REPRESENTATION

In this section we present how classical loop transformations like interchange, fission, strip mining, unroll and jam or loop blocking can be expressed as scheduling transformations. Furthermore, we show how partial transformations that happen commonly after index set splitting [25] can be expressed.

INTERCHANGE

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    Stmt(i,j);
```

$$\begin{aligned}\mathcal{D}_{Stmt} &= [N,M] \rightarrow \{Stmt[i,j] : 0 \leq i < N \wedge 0 \leq j < M\}\\ \mathcal{S}_{Stmt} &= \{Stmt[i,j] \rightarrow \Theta[i,j]\}\end{aligned}$$

A statement $Stmt$ surrounded by two loops is described by a two-dimensional domain $\mathcal{D}_{Stmt}$ and a schedule $\mathcal{S}_{Stmt}$, which maps from this domain into a two-dimensional time. To interchange the two loops we define a transformation $\mathcal{T}_{Interchange}$ that maps from the original time to a new time, with the dimensions of the new time being flipped. Applying $\mathcal{T}_{Interchange}$ on $\mathcal{S}_{Stmt}$ yields a new schedule $\mathcal{S'}_{Stmt}$. This new schedule leads to code that has interchanged loops in respect to the original code.

$$\begin{aligned}\mathcal{T}_{Interchange} &= \{\Theta[s_0,s_1] \rightarrow \Theta[s_1,s_0]\}\\ \mathcal{S'}_{Stmt} &= \mathcal{S}_{Stmt} \circ \mathcal{T}_{Interchange}\\ &= \{Stmt[i,j] \rightarrow \Theta[j,i]\}\end{aligned}$$

```
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    Stmt(i,j);
```

Following this concept an interchange of two loops in a domain with n loop dimensions can be defined or even more general, an arbitrary transposition of loop dimensions.

FISSION

```
for (i = 0; i < N; i++)
  Stmt_A(i);
  for (j = 0; j < M; j++)
    Stmt_B(i,j);
```

$$\begin{aligned}\mathcal{D}_{Stmt_A} &= [N,M] \rightarrow \{Stmt_A[i] : 0 \leq i < N\}\\ \mathcal{D}_{Stmt_B} &= [N,M] \rightarrow \{Stmt_B[i,j] : 0 \leq i < N \wedge 0 \leq j < M\}\\ \mathcal{S}_{Stmt_A} &= \{Stmt_A[i] \rightarrow \Theta[i,0,0]\}\\ \mathcal{S}_{Stmt_B} &= \{Stmt_B[i,j] \rightarrow \Theta[i,1,j]\}\end{aligned}$$

An imperfectly nested set of loops containing two statements $Stmt_A$ and $Stmt_B$ can be split into two independent loop nests. For this we introduce a new scalar time dimension that maps both statements to two hyperplanes which intersect the new dimension at distinct points. In our example this is expressed by the transformations $\mathcal{T}_{Fission_{First}}$ and $\mathcal{T}_{Fission_{Last}}$. Applying $\mathcal{T}_{Fission_{First}}$ to $\mathcal{S}_{Stmt_A}$ and $\mathcal{T}_{Fission_{Last}}$ two $\mathcal{S}_{Stmt_B}$ yields to two new schedules $\mathcal{S}'_{Stmt_A}$ and $\mathcal{S}'_{Stmt_B}$. They lead to code to execute the two statements in two distinct loop nests.

$$\begin{aligned}
\mathcal{T}_{Fission_{First}} &= \{\Theta[s_0, s_1, s_2] \rightarrow \Theta[0, s_0, s_1, s_2]\} \\
\mathcal{T}_{Fission_{Last}} &= \{\Theta[s_0, s_1, s_2] \rightarrow \Theta[1, s_0, s_1, s_2]\} \\
\mathcal{S}'_{Stmt_A} &= \mathcal{S}_{Stmt_A} \circ \mathcal{T}_{Fission_{First}} \\
&= \{Stmt_A[i] \rightarrow \Theta[0, i, 0, 0]\} \\
\mathcal{S}'_{Stmt_B} &= \mathcal{S}_{Stmt_B} \circ \mathcal{T}_{Fission_{Last}} \\
&= \{Stmt_B[i, j] \rightarrow \Theta[1, i, 1, j]\}
\end{aligned}$$

```
for (i = 0; i < N; i++)
  Stmt_A(i);

for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    Stmt_B(i,j);
```

Similarly it is possible to split loop nests starting at a loop depth of n. In this case the new scalar dimension is not inserted at the first time dimension (counting from one), but at the time dimension preceding the loop dimension that should be split.

STRIP MINING

```
for (i = 0; i < N; i++)
  Stmt(j);
```

$$\begin{aligned}
\mathcal{D}_{Stmt} &= [N] \rightarrow \{Stmt[i] : 0 \leq i < N\} \\
\mathcal{S}_{Stmt} &= \{Stmt[i] \rightarrow \Theta[i]\}
\end{aligned}$$

To execute a single loop section wise it can be transformed into a nest of two loops by applying strip mining. This transformation is not profitable on its own, but it is the base for transformations like partial unrolling, loop blocking or unroll and jam. In our example we want to execute sections of four loop iterations. Each section should be numbered by an integer t where $t \bmod 4 = 0$ holds and it should enumerate the iterations from t to $t + 3$. This is specified by transformation $\mathcal{T}_{StripMine}$. Applying $\mathcal{T}_{StripMine}$ to $\mathcal{S}_{Stmt}$ yields to the new schedule $\mathcal{S}'_{Stmt}$ and generates the expected loop nest.

$$\begin{aligned}
\mathcal{T}_{StripMine} &= \{\Theta[s_0] \rightarrow \Theta[t, s_0] && : t \bmod 4 = 0 \wedge t \leq s_0 < t + 4\} \\
\mathcal{S}'_{Stmt} &= \mathcal{S}'_{Stmt} \circ \mathcal{T}_{StripMine} \\
&= \{Stmt[s_0] \rightarrow \Theta[t, s_0] && : t \bmod 4 = 0 \wedge t \leq s_0 < t + 4\}
\end{aligned}$$

```
for (ii = 0; ii < N; ii+=4)
  for (i = ii; i < min(N, ii+4); i++)
    Stmt(i)
```

UNROLL-AND-JAM

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    Stmt(i,j)
```

$$\begin{aligned} Context &= [N, M] \to \{[] : N \bmod 4 = 0\} \\ \mathcal{D}_{Stmt} &= [N, M] \to \{Stmt[i, j] : 0 \leq i < N \wedge 0 \leq j < M\} \\ \mathcal{S}_{Stmt} &= \{Stmt[i, j] \to \Theta[i, j]\} \end{aligned}$$

Unroll-and-jam is a transformation that (partially) unrolls a non-innermost loop and then subsequently fuses the created loop nests back together. We express this as a polyhedral transformation that creates an innermost loop with a constant number of iterations. This loop can then be unrolled by a simple code generation pass or it can be transformed to SIMD code, if parallel execution is valid. In case the unroll factor does not divide the number of original loop iterations in whole numbers, some preparing transformations are necessary to apply unroll and jam.

To perform the loop transformation part of unroll and jam we create the transformations $\mathcal{T}_{StripMine}$ and $\mathcal{T}_{Interchange}$ which are conceptually already known. Combining both yields to the transformation $\mathcal{T}_{UnrollAndJam}$, which applied to $\mathcal{S}_{Stmt}$ produces the schedule $\mathcal{S}'_{Stmt}$ and generates the expected loop nest.

$$\begin{aligned} \mathcal{T}_{StripMine} &= \{\Theta[s_0, s_1] \to \Theta[t, s_0, s_1] : t \bmod 4 = 0 \wedge t \leq s_0 < t + 4\} \\ \mathcal{T}_{Interchange} &= \{\Theta[s_0, s_1, s_2] \to \Theta[s_0, s_2, s_1]\} \\ \mathcal{T}_{UnrollAndJam} &= \mathcal{T}_{StripMine} \circ \mathcal{T}_{Interchange} \\ &= \{\Theta[s_0, s_1] \to \Theta[t, s_1, s_0] : t \bmod 4 = 0 \wedge t \leq s_0 < t + 4\} \\ \mathcal{S}'_{Stmt} &= \mathcal{S}_{Stmt} \circ \mathcal{T}_{UnrollAndJam} \\ &= \{Stmt[i, j] \to \Theta[t, j, i] : t \bmod 4 = 0 \wedge t \leq i < t + 4\} \end{aligned}$$

```
for (ii = 0; ii < N; ii+=4)
  for (j = 0; j < M; j++)
    for (i = ii; i < ii+4; i++)
      Stmt(i,j);
```

LOOP BLOCKING

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    Stmt(i,j);
```

$$\begin{aligned} \mathcal{D}_{Stmt} &= [N, M] \to \{Stmt[i, j] : 0 \leq i < N \wedge 0 \leq j < M\} \\ \mathcal{S}_{Stmt} &= \{Stmt[i, j] \to \Theta[i, j]\} \end{aligned}$$

Loop blocking is a transformation that can increase data locality by executing a loop nest block wise. It can be represented by combining the transformations $\mathcal{T}_{StripMine_{Outer}}$, $\mathcal{T}_{StripMine_{Inner}}$ and $\mathcal{T}_{Interchange}$ to a single transformation $\mathcal{T}_{Block}$.

$$\begin{aligned}
\mathcal{T}_{StripMine_{Outer}} &= \{\Theta[s_0,s_1] \to \Theta[t,s_0,s_1] : t \bmod 4 = 0 \land t \leq s_0 \leq t+3\} \\
\mathcal{T}_{StripMine_{Inner}} &= \{\Theta[s_0,s_1,s_2] \to \Theta[s_0,s_1,t,s_2] : t \bmod 4 = 0 \land t \leq s_2 \leq t+3\} \\
\mathcal{T}_{Interchange} &= \{\Theta[s_0,s_1,s_2,s_3] \to \Theta[s_0,s_2,s_1,s_3]\} \\
\mathcal{T}_{Block} &= \mathcal{T}_{StripMine_{Outer}} \circ \mathcal{T}_{StripMine_{Inner}} \circ \mathcal{T}_{Interchange} \\
&= \{\Theta[s_0,s_1] \to \Theta[t_0,t_1,s_0,s_1] : t_0 \bmod 4 = 0 \land t_0 \leq s_0 < t_0+4 \\
&\qquad \land\ t_1 \bmod 4 = 0 \land t_1 \leq s_1 < t_1+4\} \\
\mathcal{S}'_{Stmt} &= \mathcal{S}_{Stmt} \circ \mathcal{T}_{Block} \\
&= \{Stmt[i,j] \to \Theta[t_i,t_j,i,j] : t_i \bmod 4 = 0 \land t_i \leq i < t_i+4 \\
&\qquad \land\ t_j \bmod 4 = 0 \land t_j \leq j < t_j+4\}
\end{aligned}$$

```
for (ii = 0; ii < M; ii+=4)
  for (jj = 0; jj < N; jj+=4)
    for (i = ii; i < min(M, ii+4); i++)
      for (j = jj; j < min(N, jj+4); j++)
        Stmt(i,j);
```

PARTIAL TRANSFORMATIONS - INDEX SET SPLITTING

```
for (i = 0; i <= 2 * N; i++)
  A[i] = A[2 * N - 1];  // Stmt
```

$$\begin{aligned}
\mathcal{D}_{Stmt} &= [N] \to \{Stmt[i,j] : 0 \leq i \leq N\} \\
\mathcal{S}_{Stmt} &= \{Stmt[i] \to \Theta[i]\}
\end{aligned}$$

In some cases it is beneficial to split the iteration domain of a statement and to create for different parts of the iteration domain different schedules. Such a transformation is known as index set splitting as presented by Griebl et al. [25]. To represent it in Polly, there is no need to create two separate statements. Instead we define a transformation that applies only to a part of the index space. We call such a transformation partial transformation.

In our example we have a single loop that cannot be executed in parallel because of dependencies between the lower and the upper half of the iteration space. However, if we apply loop fission and split the lower and the upper part of the iteration space into two separate loops, two fully parallel loops can be generated. To describe this we define two transformations, that generate a new time dimension on which we split the iterations. $\mathcal{T}_{Fission_{First}}$ maps the lower part of the iteration space to an earlier time in this dimension, and $\mathcal{T}_{Fission_{Last}}$ maps the upper part of the iteration space to a later time in this dimension. Both can be merged to a single transformation $\mathcal{T}_{Fission_{Combined}}$, which is subsequently applied to $\mathcal{S}_{Stmt}$. This results in the new schedule $\mathcal{S}'_{Stmt}$ which produces the expected parallel loops.

$$\begin{aligned}
\mathcal{T}_{Fission_{First}} &= [N] \rightarrow \{\Theta[s_0] \rightarrow \Theta[0, s_0] : s_0 \leq N\} \\
\mathcal{T}_{Fission_{Last}} &= [N] \rightarrow \{\Theta[s_0] \rightarrow \Theta[1, s_0] : s_0 > N\} \\
\mathcal{T}_{Fission_{Combined}} &= \mathcal{T}_{Fission_{First}} \cup \mathcal{T}_{Fission_{Last}} \\
&= [N] \rightarrow \{\Theta[s_0] \rightarrow \Theta[0, s_0] : s_0 \leq N; \Theta[s_0] \rightarrow \Theta[1, s_0] : s_0 > N\} \\
\mathcal{S}'_{Stmt} &= \mathcal{S}_{Stmt} \circ \mathcal{T}_{Fission_{Combined}} \\
&= [N] \rightarrow \{Stmt[i] \rightarrow \Theta[0, i] : i \leq N; \Theta[i] \rightarrow \Theta[1, i] : i > N\}
\end{aligned}$$

```
# parallel
for (i = 0; i <= N; i++)
  A[i] = A[2 * N - 1];   // Stmt

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

6.2 EXTERNAL OPTIMIZERS - JSCOP

Polly can export its internal polyhedral representation and allows to reimport an optimized version of it. As a result, new optimizations can be tested without any knowledge about compiler internals. It is sufficient to analyze and optimize an abstract polyhedral description. This facility can be used to try optimizations manually, but also to connect existing optimizers with Polly or to develop new research prototypes.

Polly supports two exchange formats. The first one is the scoplib format[1], as currently used by Clan, Candl and Pluto [13]. The second one is called *JSCoP* and is specific to Polly. The main reason for defining our own exchange format is that scoplib is not expressive enough for Polly's internal representation. Scoplib does not allow partial schedules or memory accesses that touch more than one element at a time, and it cannot represent existentially quantified variables. In case such constructs appear in the description of a SCoP, it is not possible to derive a valid scoplib file. JSCoP is a file format based on JSON [17]. It contains a polyhedral description of a SCoP, that follows the one used in scoplib, but it is more general. In case a tool wants to use JSCoP, but works with a less generic definition, we propose to detect and ignore unsupported SCoPs or to take conservative assumption. To read or write JSCoP files one of the many available JSON libraries can be used.

A JSCoP is defined based on JSON elements. There are three types of elements used. The first one is a string of characters written as *"sometext"*, the second one is a list of elements $[e_0, e_1, ..., e_n]$ and the third one is a map $\{k_0 : e_0, k_1 : e_1, ..., k_n : e_n\}$, that maps from strings to elements. A JSCoP file may also contain integer sets and maps. They are stored as strings using the textual representation of isl.

The root element in the JSCoP file describes a single SCoP. It is a map containing the three elements `name`, `context` and `statements`. `name` is a string defining the name of the

---

[1] http://www.cse.ohio-state.edu/~pouchet/software/pocc/download/modules/scoplib-0.2.0.tar.gz

```
{
  "name": "body => loop.end",
  "context": "[N] -> { []: N >= 0 }",
  "statements": [{
    "name": "Stmt_body",
    "domain": "[N] -> { Stmt_body[i0, i1, i2] : 0 <= i0, i1, i2 <= N }",
    "schedule": "[N] -> { Stmt_body[i0, i1, i2] -> scattering[i0, i1, i2] }",
    "accesses": [{
      "kind": "read",
      "relation": "[N] -> { Stmt_body[i0, i1, i2] -> MemRef_C[i0][i1] }"
    },
    {
      "kind": "read",
      "relation": "[N] -> { Stmt_body[i0, i1, i2] -> MemRef_A[i0][i2] }"
    },
    {
      "kind": "read",
      "relation": "[N] -> { Stmt_body[i0, i1, i2] -> MemRef_B[i1][i2] }"
    },
    {
      "kind": "write",
      "relation": "[N] -> { Stmt_body[i0, i1, i2] -> MemRef_C[i0][i1] }"
    }]
  }]
}
```

Figure 19: *JSCoP file of a matrix multiplication kernel*

SCoP, `context` is an integer set describing the set of valid parameters and `statements` is a list of the statements as subsequently defined. `context` has zero set dimensions and as many parameter dimensions as there are parameters in the SCoP.

A statement is a map containing four elements called `name`, `domain`, `schedule` and `accesses`. `name` is a unique string that identifies a statement and `domain` is the iteration space of the statement. The iteration space describes the set of statement instances that will be executed. It is represented as an integer set, which has as many parameter dimensions as the context of the SCoP and as many set set dimensions as there are loops that surround the statement. The space defined by `domain` is named with the name of the statement. `schedule` is a map that defines for each statement instance a multi-dimensional execution time. It has as many input dimensions as `domain` has set dimension and an arbitrary number of output dimensions Yet, all schedules in the SCoP must have the same number of output dimensions. The name of the output space is 'scattering'. `accesses` is a list of the memory accesses of the statement.

A memory access is a map containing two elements called `type` and `relation`. The `type` can be `read`, `write` or `maywrite`. It is `read` if the statement must or may read this memory. It is `write` if the statement must write to this memory and it is `maywrite` if the statement may or may not write to this memory. If a statement reads and writes identical memory, two accesses are generated, one for the read and one for the write access. `relation` defines the memory accessed. It is a mapping from the statement instance executed to the memory element accessed. The input space of the access relation has the same number of dimensions as the domain of the statement and is also identically named. The output space defines the array elements accessed. It has a unique name for each distinct array accessed in the SCoP and describes a possibly multi-dimensional space of array elements.

All data structures are read-only, except the schedules of the statements. All transformations are expressed by modifying the schedules of the statements. To support array expansion or memory access transformations we plan to also allow updates of the access relations. Figure 19 shows an example of a JSCoP file of a matrix multiplication kernel.

6.3 DEPENDENCY ANALYSIS

Polly provides an advanced dependency analysis implemented on top of the isl provided data flow analysis. The isl data flow analysis is based on techniques developed by Feautrier [20] with influences from Pugh and Wonnacott [39]. It includes full support for existentially quantified variables, can be restricted to non-transitive dependences and allows full access relations as well as may-write accesses. We believe the last two features do no exist in most implementations of data-dependency analyses.

Polly currently uses the dependency analysis to calculate read-after-write, write-after-write and write-after-read dependences. The dependences are calculated from the polyhedral description. We pass the access relations and types, the domains and the schedules of all statements in the SCoP to the isl data flow analysis. isl then calculates exact, non-transitive data flow dependences as well as the statement instances that do not have any source and consequently depend on the state the memory has before the execution of the SCoP.

The following code is a simple matrix multiply kernel:

```
for (i = 0; i < 100; i++)
  for (j = 0; j < 100; j++)
    for (k = 0; k < 100; k++)
        C[i][j] += A[i][k] + B[k][j];
```

$$\begin{aligned}
\mathcal{D} &= \{Stmt[i,j,k] : 0 \leq i,j,k < 100\} \\
\mathcal{A}_{ReadA} &= \{Stmt[i,j,k] \rightarrow A[i,k]\} \\
\mathcal{A}_{ReadB} &= \{Stmt[i,j,k] \rightarrow B[k,j]\} \\
\mathcal{A}_{ReadC} &= \{Stmt[i,j,k] \rightarrow C[i,j]\} \\
\mathcal{A}_{WriteC} &= \{Stmt[i,j,k] \rightarrow C[i,j]\} \\
\mathcal{S} &= \{Stmt[i,j,k] \rightarrow [i,j,k]\}
\end{aligned}$$

Our dependency analysis returns for this kernel a single read-after-write dependency $\{Stmt_C[i,j,k] \rightarrow Stmt_C[i,j,k+1] : 0 \leq i,j < 100 \wedge 0 \leq k < 99\}$. The dependency only exists from one iteration to the immediately following iteration. Even though iteration $[0,0,0]$ writes into $C[0][0]$ and iteration $[0,0,10]$ reads from the same element $C[0][0]$, this dependency is not reflected. The reason is that there exist intermediate writes for example at $[0,0,9]$ that overwrite the value of $C[0][0]$. Dependences that are hidden by such intermediate writes are not calculated to keep the number of dependences low and to allow maximal flexibility in terms of possible code transformations. Another interesting point is that the iterations $[*,*,0]$ are not target of any dependency. This is because the initial write into C is not part of the SCoP. The same is true for A and B which are also not initialized in the SCoP. Hence the dependency analysis provides additionally the following set of statement-access combinations that must not have any source $\{Stmt_C[i,j,0] : 0 \leq i,j < 100; Stmt_A[i,j,k] : 0 \leq i,j,k < 100; Stmt_B[i,j,k] : 0 \leq i,j,k < 100;\}$. For this kernel the analyses calculates write-after-write and write-after-read dependences with their corresponding no-source-sets that are similar to the previously shown read-after-write dependences.

The example we have seen, did not contain any may-write accesses and Polly itself does currently neither generate may-write accesses nor does it create accesses to ranges of memory. Still, we want to explain these features, as they are supported by the isl dependency analysis and they will be soon integrated in Polly. Looking at Figure 20 we see that StmtTwo has an array subscript that is calculated by a function call to $foo()$. The actual function calculated by $foo()$ is unknown such that it is impossible to derive an affine access function. However, if we assume $foo()$ is side-effect free, we can conservatively assume that in StmtTwo the whole array C may be written. We model this as the may-write access $\mathcal{A}_{MayWriteStmtTwo} = \{StmtTwo[i] \rightarrow A[o]\}$ that accesses an unlimited range of memory (There are no restrictions on o). The dependency analysis can now calculate the write accesses that have dependences to the read in StmtThree. The result of this calculation are two maps. One for the must-dependences and another for the may-dependences. The must-dependences are $\{StmtOne[i] \rightarrow StmtThree[i] : 50 \leq i < 100\}$ as for $i >= 50$ the only write that could have defined the value of $C[i]$ is the one at StmtOne. StmtTwo is for $i >= 50$ not executed. The may-dependences are $\{StmtOne[i] \rightarrow$

```
for (i = 0; i < 100; i++)
  C[i] = 0;                    // StmtOne

  for (j = i; j < 50; j++)
    C[foo(j)] = j;             // StmtTwo

  ... = C[i];                  // StmtThree
```

$$
\begin{aligned}
\mathcal{D}_{StmtOne} &= \{StmtOne[i] : 0 \leq i < 100\} \\
\mathcal{D}_{StmtTwo} &= \{StmtTwo[i,j] : 0 \leq i < 100 \wedge i \leq j < 50\} \\
\mathcal{D}_{StmtThree} &= \{StmtThree[i] : 0 \leq i < 100\} \\
\mathcal{A}_{WriteStmtOne} &= \{StmtOne[i] \to A[i]\} \\
\mathcal{A}_{MayWriteStmtTwo} &= \{StmtTwo[i] \to A[o]\} \\
\mathcal{A}_{ReadStmtThree} &= \{StmtThree[i] \to A[i]\} \\
\mathcal{S}_{StmtOne} &= \{StmtOne[i] \to [i,0]\} \\
\mathcal{S}_{StmtTwo} &= \{StmtTwo[i,j] \to [i,1,j]\} \\
\mathcal{S}_{StmtThree} &= \{StmtThree[i] \to [i,2]\}
\end{aligned}
$$

Figure 20: *MayWrites and accesses to memory ranges*

$StmtThree[i] : 0 \leq i < 50; StmtOne[i] \to StmtTwo[i,j] : 0 \leq i < 100 \wedge i \leq j < 50\}$ as for these statement instances it is unknown which dependency will at runtime define the value that is actually read. By using two types of dependences we can provide conservative may-dependences if approximations are needed, but can use exact must-dependences in all possible cases. This ensures a high SCoP coverage and simultaneously maximal flexibility in terms of transformations.

7

POLYHEDRAL DESCRIPTION TO LLVM-IR

In Polly transformations are applied on the polyhedral description of a SCoP. The LLVM-IR of a program remains unchanged throughout a longer sequence of transformations. Only after of all transformations have been applied on the polyhedral description, LLVM-IR reflecting this possibly changed description is regenerate.

This section describes in two parts how we regenerate LLVM-IR. The first part describes how we retrieve a generic AST that describes a program as defined in the polyhedral representation. The second part describes how to use this AST to generate optimized LLVM-IR. We discuss the generation of sequential code, but also techniques to detect parallelism and to generate optimized OpenMP or SIMD code.

7.1 GENERATION OF A GENERIC AST

The first step from the polyhedral representation of a SCoP back to a program described in LLVM-IR is to generate a loop structure which enumerates all statement instances in the order described by the schedule. This loop structure is first described by a generic AST that is not yet related to LLVM-IR.

Bastoul [11] developed with the Chunky Loop Generator (CLooG) an efficient code generator, that solves exactly this problem. It uses an improved version version an algorithm introduced by Quilleré et al. [41]. Further improvements that include for example support for existentially quantified variables, partial schedules as well as code generation that uses exact $\mathbb{Z}$-polyhedra were introduced by Verdoolaege [50] who also contributed the CLooG isl backend.

Polly itself offloads the calculation of the generic AST completely to CLooG.

7.2 ANALYSES ON THE GENERIC AST

For some cases analyses on the generic AST are necessary to derive detailed information. The polyhedral representation defines the execution order of the statement instances in a SCoP. Yet, it does not define the exact control flow structures used to ensure this execution order. Different ASTs can be generated for the same SCoP depending on the implementation of the code generator or the chosen optimization goals. Listing 1 shows two different ASTs (represented as C code), that are both generated from the same SCoP. The first one has minimal code size, as it does not contain duplicated statements. The second one has minimal branching with all conditions removed from the loop bodies.

```
/* SCoP 1 */
for (i = 0; i <= N; i++) {           // Loop $L_1$
  if (i <= 10)
    B[0] += i;

  A[i] = i;
}

/* SCoP 2 */
for (i = 0; i <= min(10, N); i++) { // Loop $L_{2.1}$
  B[0] += i;
  A[i] = i;
}

for (i = 11; i <= N; i++)            // Loop $L_{2.2}$
  A[i] = i;
```

Listing 1: *Two different ASTs generated from the same SCoP.*

In general, an analysis that derives information on the created AST is imprecise if performed before code generation. Decisions taken during code generation are not available to an early analysis such that always the complete scattering space needs to be considered instead of the actual subspaces enumerated by the individual loops. Loop $L_{2.2}$ in Listing 1 can for example be executed in parallel, but an analysis that does not take code generation into account cannot derive this.

The code in Listing 1 is generated from the following SCoP:

$$\begin{aligned}
\mathcal{D}_A &= [N] \to \{A[i] : 0 \leq i \leq N\} \\
\mathcal{S}_A &= [N] \to \{A[i] \to \mathit{Scattering}[i]\} \\
\mathcal{D}_B &= [N] \to \{B[i] : 0 \leq i \leq 10\} \\
\mathcal{S}_B &= [N] \to \{B[i] \to \mathit{Scattering}[i]\}
\end{aligned}$$

To perform a precise analysis we extract information on the subset of the scattering space that is enumerated by each loop and include it in the analysis we perform. For example for the loops in Listing 1 the following information is added:

$$\begin{aligned}
\mathcal{E}_{L_1} &= [N] \to \{\mathit{Scattering}[i] : 0 \leq i \leq N\} \\
\mathcal{E}_{L_{2.1}} &= [N] \to \{\mathit{Scattering}[i] : 0 \leq i \wedge i \leq N \wedge i \leq 10\} \\
\mathcal{E}_{L_{2.2}} &= [N] \to \{\mathit{Scattering}[i] : 11 \leq i \leq N\}
\end{aligned}$$

Even though this information is obtained after code generation, it is not obtained by reparsing the generated code. Instead, the polyhedral information is exported directly from the code generator. This is different to approaches taken for example in PoCC where the AST is reparsed to derive the necessary polyhedral information.

7.2.1 *Detection of Parallel Loops*

Polly detects loops that can be executed in parallel to automatically generate OpenMP or SIMD code. As explained in the previous section this analysis is performed after code generation, to detect all forms of parallelism. This includes parallelism that existed in the original code, parallelism that was exposed by transformations through a changed schedule and parallelism that was introduced during generation of the AST. Even though external optimizers could provide information on parallelism, Polly does intentionally not rely on any external information. As a result Polly can ensure that all introduced parallelism maintains the correctness of the program.

Listings 2 and 3 show code where the instances of two statements are either enumerated in a single loop nest or in two separate ones. Which code is generated depends on the schedule of the statements. In case of a single loop nest no loop can be executed in parallel, as the i- and j loop carry dependences of $S1$ and the k-loop carries dependences of $S2$. However, in the case of two separate loop nests, each loop nests contains loops that do not carry any dependency. In the first loop nest these are the i- and j-loop and in the second one this is the k-loop. All loops that do not carry any dependency can be executed in parallel.

To decide if a certain loop, e.g. the innermost loop in the second loop nest of Listing 3, is parallel we check if it carries any dependences. We start from the set of dependences as calculated in Chapter 6.3. For our example these are described by the union map $\mathcal{D} = \{S1[i,j,k] \rightarrow S1[i,j,k+1];\ S2[i,j,k] \rightarrow S2[i,j+1,k]; S2[i,-1+N,k] \rightarrow S2[1+i,0,k]\}$. $\mathcal{D}$ contains relations between statement instances. We translate them into the scheduling space by applying on their ranges and domains the statement schedules contained in $\mathcal{S} = \{S1[i,j,k] \rightarrow S1[0,i,j,k]; S2[i,j,k] \rightarrow [1,i,j,k]\}$. The resulting dependences are described by the union map $\mathcal{D}_S = \mathcal{S}^{-1}.\mathcal{D}.\mathcal{S} = \{[0,i,j,k] \rightarrow [0,i,j,k+1];\ [1,i,j,k] \rightarrow [1,i,j+1,k]; [1,i,-1+N,k] \rightarrow [1,1+i,0,k]\}$. $\mathcal{D}_S$ is now limited to the dependences in the second loop nest by intersecting its domains and ranges with the scheduling space enumerated in this loop nest. The space enumerated is $\mathcal{E}_{L2} = [M,N,K] \rightarrow \{[1,i,j,k] : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$ such that the remaining dependences are $\mathcal{D}_{L2} = \{[1,i,j,k] \rightarrow [1,i,j+1,k]; [1,i,-1+N,k] \rightarrow [1,1+i,0,k]\}$. If we now calculate the dependence distances with $deltas(\mathcal{D}_{L2}) = \{[0,0,1,0]; [0,1,1-N,0]\}$, we can see that the second and the third dimension carry dependences while the others are parallel. As the very first dimension is a scalar dimension only the innermost dimension and consequently the innermost loop is parallel. If we would have checked the innermost dimension before code generation, it would have carried a dependency and consequently parallel execution would have been invalid. Though after code generation, we were able to eliminate dependences in certain loop nests. Proving that the innermost dimension of the second loop nest is parallel became possible.

7.2.2 *The Stride of a Memory Access Relation*

Especially for SIMD code generation it is important to understand the pattern in which memory is accessed during the execution of a loop. For common pattern special vector instructions can be used to load or store memory in a more efficient way. To detect such pattern we calculate the distance between the memory accesses of two subsequently ex-

```
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++) {
      C[i][j] += k; // S1
      X[k] += k;    // S2
    }
```

Listing 2: *A single loop nest hides parallelism*

```
for (i = 0; i < M; i++)                         // parallel loop
  for (j = 0; j < N; j++)                       // parallel loop
    for (k = 0; k < K; k++)
      C[i][j] += k;  // S1

for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)                     // parallel loop
      X[k] += k;     // S2
```

Listing 3: *Two separate loop nests expose parallelism*

ecuted statement instances. This distance is called the stride of the memory accesses relation and is calculated in respect to the innermost loop. The analysis used is again fully polyhedral. For accurate results every loop in the generated AST is analyzed individually, such that the iteration space taken into account is limited to the subset enumerated by each loop.

To calculate for a statement **S** the stride of a specific memory access **X** the following information is needed: $\mathcal{A}_X$, the access relation to array 'X' , $\mathcal{S}$ the scattering function of the statement containing the memory access and $\mathcal{E}$, the subset of the scattering space enumerated by the loop we analyze.

We explain this with the following example:

```
for (i = 0; i <= N; i++)
  for (j = 0; j <= N; j++) {
    A[j] =  B[i] + C[2 * j];
```

and its polyhedral representation:

$$\mathcal{A}_A = [N] \rightarrow \{Stmt[i,j] \rightarrow A[j]\} \tag{7.1}$$

$$\mathcal{A}_B = [N] \rightarrow \{Stmt[i,j] \rightarrow B[i]\} \tag{7.2}$$

$$\mathcal{A}_C = [N] \rightarrow \{Stmt[i,j] \rightarrow C[2j]\} \tag{7.3}$$

$$\mathcal{S} = [N] \rightarrow \{Stmt[i,j] \rightarrow Scattering[i,j]\} \tag{7.4}$$

$$\mathcal{E} = [N] \rightarrow \{Scattering[s_0, s_1] : 0 \leq s_0 \leq N \wedge 0 \leq s_1 \leq N\} \tag{7.5}$$

The first step is to create a map from one iteration to the next iteration. We start with a map that maps from the scattering space back into the scattering space. Then we add constraints that force input and output to have identical values for all dimensions smaller than the dimension of the loop that is analyzed. At the dimension of the loop we analyze the input has to be smaller than the output. This is enforced by the following map:

$$LT = [N] \rightarrow \{Scattering[i_0, i_1] \rightarrow Scattering[o_0, o_1] : i_0 = o_0 \wedge i_1 < o_1\}$$

LT is then limited to the elements that are actually enumerated by our loop. For this we intersect the range and the domain of LT with S_{actual}. The result is called $LT_{scattering}$.

$$LT_{scattering} = S_{actual} \cap LT \cap S_{actual}$$

$LT_{scattering}$ is used to obtain for each point in the scattering space the point that is executed subsequently. To get this point we calculate the lexicographic minimum of $LT_{scattering}$. The result is called $NEXT$. $NEXT$ is still a mapping from one scattering point to another. To create a mapping between different statement instances we apply the scattering on both sides, range and domain, of $NEXT$.

$$NEXT = lexmin(LT_{scattering})$$
$$NEXT_{Dom} = S^{-1}.NEXT.S$$

With $NEXT_{Dom}$ we can now create a map that gives for each memory address accessed the memory address that is accessed by the subsequently executed statement instance. To create this map we apply the access maps on both sides of $NEXT_{Dom}$.

$$NEXT_{Access_X} = A_X^{-1}.NEXT_{Dom}.A_X$$

The last step to obtain the stride of subsequent memory accesses is to calculate the distance between the access on the left side and the one on the right side.

$$Stride_X = delta(Next_{Access_X})$$

There exists a large variety of possible strides, however some common ones are worth mentioning. A stride of zero means that the memory accessed during the execution of a loop is always at the same location. It is also called a constant stride memory access. A statement with a stride-one memory access always accesses subsequent memory cells in increasing order. A statement with a stride equal to minus one accesses subsequent memory cells in decreasing order.

7.3 GENERATION OF LLVM-IR

To generate optimized LLVM-IR for the program, we use the generic AST as a description of the final loop structure. Based on this description we create the matching LLVM-IR instructions. Polly provides three ways to generate LLVM-IR. Sequential code generation, OpenMP code generation and SIMD code generation. Their only difference is the way loops are code generated.

7.3.1 *Sequential Code Generation*

Polly generates by default sequential code. For this it replaces each abstract construct in the generic AST with a sequential LLVM-IR implementation. The control flow structures are generated by replacing the abstract for-loops with LLVM-IR loops and abstract if-conditions with LLVM-IR conditional branches. The actual calculations are reintroduced by replacing each abstract statement with the corresponding LLVM-IR basic block. The general code generation is straightforward, such that implementation details are not explained here. The interested reader may take a look at the source code documentation. Yet, there are two points worth mentioning. The benefits of independent blocks for code generation and the problem of deriving a correctly sized type for newly introduced variables.

As stated in Section 5.6.2 we require that all basic blocks in a SCoP are independent basic blocks that only reference induction variables or parameters. This allows a very simple code generation algorithm. First of all we create code that calculates the value of all parameters that are not just a single register, but constant subexpressions which we were not able to analyze further. The newly calculated parameter registers, are now stored in a map which is later used to replace in each basic block references to old parameters with references to new ones. To obtain the new values of the induction variables a similar approach is taken. CLooG provides in its generated generic AST for each original induction variable an expression that describes it in terms of new induction variables and parameters. We take this expression, translate it into LLVM-IR and replace all occurrences of the old induction variables in a basic block with their newly calculated values. The translated basic block can now easily be inserted into the regenerated control flow structures. Then we only need to remove the old induction variables and branch instructions to obtain a working program. There is no need to insert any PHI-instructions, as the only PHI-instructions allowed in the input SCoP were induction variables. We also do not spend any afford on optimizing possibly redundant scalar computations, as they are for example introduced by the independent blocks pass. Here, we rely entirely on the standard LLVM optimization passes to remove such computations. As shown in Chapter 9, this works very reliable.

The next interesting problem is the selection of the data types for the newly generated induction variables and array subscripts. At the moment Polly uses always 64 bit induction variables and signed calculations. This is correct as long as the input loops use signed calculations for their induction variables (ensured in the frontend), these variables have at most 64 bit size and the schedules of the statements do not produce any larger values. A schedule that may produce larger values is for instance $\{Stmt[i,j] \rightarrow [i+j]\}$. It may produce an overflow when calculating the bounds of $i+j$. However, in practical programs we have not yet seen such an overflow as most loops do not get close to the maximal value possible in a 64 bit counter. To ensure correctness even in uncommon pathological cases and to get native types for non 64 bit architectures, we plan to automatically derive the minimal and maximal values the newly generated induction variables may have. For this we plan to add additional information to the polyhedral representation, that defines the maximal values induction variables may reach due to their integer types in the original program. This information is then available after code generation through a polyhedra interface, such that we can conveniently derive the minimal type, needed to ensure that no unforeseen integer overflows occur. The implementation of this feature is not yet finished, as we still need to investigate the compile time overhead introduced and we need to ensure, that no no unnecessary computations are introduced into LLVM-IR because of additional constraints that CLooG cannot remove.

7.3.2 *OpenMP Code Generation*

Polly supports OpenMP code generation[1] to take advantage of shared memory parallelism in a SCoP. In case OpenMP code generation is enabled each loop that is parallel as detected by the analysis in Section 7.2.1 and not surrounded by any other parallel loop is code generated as an OpenMP parallel loop. As the run time functions are targeted the user can use environment variables to define the scheduling policy and the number of threads used for execution.

Currently only the GNU OpenMP [22] run time library is targeted, however similar libraries like the multiprocessor computing framework (MPC) developed by Perache et al. [36] can be integrated.

A detailed explanation of how OpenMP code is generated will be included in the not yet published masters thesis of Raghesh Aloor from IIT Madras.

7.3.3 *Vector Code Generation*

Polly generates vector code[2] for so-called trivially vectorizable loops. Instead of generating an explicit loop that enumerates all statement instances, only a single basic block is generated that executes all statement instances of the loop in parallel by taking advantage of LLVM vector instructions. The vector code generation in Polly is on purpose limited to a very restricted set of loops to decouple the actual vector code generation from the transformations that enable it. Therefore, previous optimizations are required to create those trivially vectorizable loops, which the Polly code generation will automatically detect and transform into efficient vector operations. Later the LLVM back ends translate the LLVM vector instructions into efficient platform specific operations.

A loop is called trivially vectorizable if it fulfills the following requirements: First it does not carry any dependency, such that it can be executed in parallel. Next it has a constant, non parametric number of loop iterations, which will be the width of the generated LLVM vector instructions. And finally the loop is an innermost loop that does not contain any conditional control flow. Listing 4 shows an example of a loop nest that is strip mined, to expose a trivially vectorizable loop, and than vectorized.

Three steps are required to detect if a loop is trivially vectorizable: First it is checked by using an AST walk, if no other loop or conditional AST node is a descendant of the current loop. If this is true the current loop is an innermost loop, free of any inner control flow. Than the number of loop iterations is calculated using the polyhedral set difference on the lower and upper loop bound. It needs to be constant and a valid vector size. Finally it is verified that the loop can be executed in parallel using the analysis described in Section 7.2.1.

---

[1] Enabled by the command line option `-enable-polly-openmp`

[2] Enabled by the command line option `-enable-polly-vector`

```
for (i = 0; i <= 1024; i++)
    B[i] = A[i];

for (i = 0; i <= 1024; i+=4)
    for (ii = i; ii <= i + 3; ii++)
        B[ii] = A[ii];

for (i = 0; i <= 1024; i+=4)
    B[i][i:(i+3)] = A[i][i:(i+3)];
```

Listing 4: *Two steps to vectorize a loop*

In the case of classical, non vector code generation control flow for a loop is created and the LLVM-IR of the original statement is embedded inside this loop. To generate vector code no loop structure is generated at all, but for each instruction in the original code a set of dedicated instructions is created, each representing a different loop iteration. All references to the current loop induction variable are replaced by references to variables, that contain the value of the induction variable at the specific loop iteration. Those instantiated loop induction variables are code generated at the beginning of the new basic block, as well as variables of the lower loop bound and the stride needed to calculate the induction variables.

Vector code is introduced as soon as a load instruction is code generated. Besides generating the scalar loads of all instances of the load instruction, vector instructions are generated that store those scalars next to each other in a vector. Any later instruction that used the loaded scalar in the original loop body is now translated into a vector instruction, such that all instances of the original instruction are executed in a single vector instruction. Starting from a load instruction, all calculations depending on the loaded value are done on vector types up to the final store operations. For the final stores the scalar values are again extracted from the vectors and stored separately.

```
for (i = 0; i < 2; i++)
  C[i] = A[i] + B[0];
```

Listing 5: *Example of a trivially vectorizable loop*

Listing 5 shows a loop that has two loop iterations, no inner control flow and can be executed in parallel. It is therefore trivially vectorizable. Vector code as shown in 7 can therefore be created from the LLVM-IR of the loop body in listing 6. At the beginning of the basic block variables for the loop stride, the lower bound of the loop as well as for each instance of the loop induction variable are generated. Furthermore for each load in the original code corresponding instructions are generated to load all accessed values into vectors `%A.vector.1` and `%B.vector.1`. The `add` instruction is code generated as a two-element vector instruction as it depends on the values loaded from `@A` and `@B`. Finally the scalar values of the `%result` are extracted and stored separately to memory.

```
%A.pointer = getelementptr [1024 x float]* @A, i64 0, i64 %indVar
%B.pointer = getelementptr [1024 x float]* @B, i64 0, i64 0
%A.scalar = load float* %A.pointer
%B.scalar = load float* %B.pointer
%result = add float %A.scalar, %B.scalar
%C.pointer = getelementptr [1024 x float]* @C, i64 0, i64 %indVar
store float %result, float* %C.pointer
```

Listing 6: *The LLVM-IR of the loop body in listing 5*

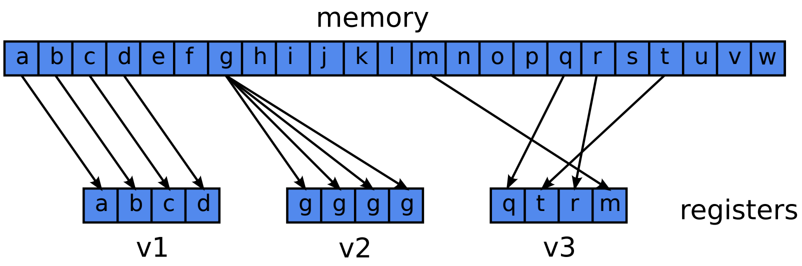


Figure 21: *Three kinds of vector loads: v1 is a stride-one vector load, v2 is a stride-zero vector load and v3 is a complex vector load*

Through the introduction of vector code several instructions can be merged into single vector operations, however the current approach still shows a large overhead for loads and stores, as each scalar is loaded or stored separately. Fortunately several common cases can be optimized. By using the analysis described in Section 7.2.2 the strides of the memory accesses are detected. Vector loads can be grouped into stride-one vector loads, stride-zero vector loads and other more complex vector loads. Stores can be grouped in either stride-one vector stores or other more complex stores. Stride zero stores do not exist in a parallel loop, as the different loop iterations would overwrite each other. Figures 21 and 22 show the different groups graphically.

All except the more complex vector accesses are optimized in Polly. A stride zero vector load, appears in the case of a load from an address that is constant during loop execution. It is optimized by loading the constant value once into a scalar and using

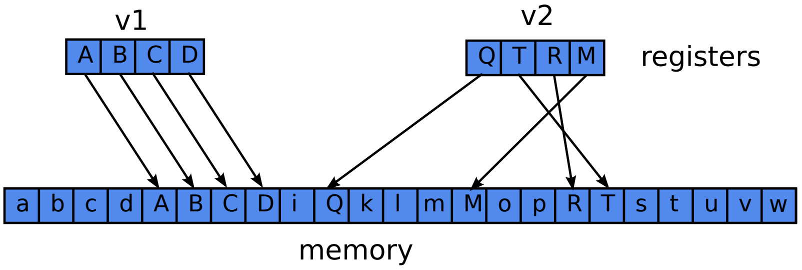


Figure 22: *Two kind of vector stores: v1 is a stride-one vector store and v3 is a complex vector store*

```
%stride = add i64 0, 1
%lowerBound = add i64 0, 0
%indVar.0 = add i64 %lowerBound, 0
%indVar.1 = add i64 %indVar.0, %stride
%A.pointer.0 = getelementptr [1024 x float]* @B, i64 0, i64 %indVar.0
%A.pointer.1 = getelementptr [1024 x float]* @B, i64 0, i64 %indVar.1
%A.scalar.0 = load float* %A.pointer.0
%A.vector.0 = insertelement <2 x float> undef, float %scalar.0, i32 0
%A.scalar.1 = load float* %A.pointer.1
%A.vector.1 = insertelement <2 x float> %vector.0, float %scalar.1, i32 1
%B.pointer.0 = getelementptr [1024 x float]* @B, i64 0, i64 0
%B.pointer.1 = getelementptr [1024 x float]* @B, i64 0, i64 0
%B.scalar.0 = load float* %B.pointer.0
%B.vector.0 = insertelement <2 x float> undef, float %scalar.0, i32 0
%B.scalar.1 = load float* %B.pointer.1
%B.vector.1 = insertelement <2 x float> %vector.0, float %scalar.1, i32 1
%result = add <2 x float> %A.vector.1, %B.vector.1
%C.pointer.0 = getelementptr [1024 x float]* @C, i64 0, i64 %indVar.0
%C.pointer.1 = getelementptr [1024 x float]* @C, i64 0, i64 %indVar.1
%C.scalar.0 = extractelement <2 x float> %result, i32 0
store float %C.scalar.0, float* %C.pointer.0
%C.scalar.1 = extractelement <2 x float> %result, i32 1
store float %C.scalar.1, float* %C.pointer.1
```

Listing 7: *Unoptimized vector code for listing 5*

a vector instruction to splat the value into a wider vector. A stride-one vector load or store, appears in the case of a memory reference accessing consecutive memory cells in subsequent loop iterations. It is optimized by loading or storing the whole vector at once using a single vector load or vector store.

Listing 8 shows the optimized vector code for the example of listing 5. The load from A is a stride-one access, that is translated to a full vector load. The load from B is a stride-zero access, that is translated to a scalar load followed by a splat, and the store to C is a stride-one access translated to a full vector store. It can be seen that the number of instructions for a memory access is reduced and is now independent of the vector width.

Future optimizations are possible starting from stride minus one, which can be optimized as a vector load plus a reversal, up to more sophisticated approaches that take more than one loop dimension into account to take advantage of possible vector loads in outer loops, that can be used to construct the vectors used in the inner loop.

To enable the LLVM back ends to generate optimal vector code it is necessary to derive the alignment of the different load and store instructions such that operations working on aligned memory can be used. However, it became apparent that LLVM can often calculate this information in its standard optimization passes.

```
%stride = add i64 0, 1
%lowerBound = add i64 0, 0
%indVar.0 = add i64 %lowerBound, 0
%A.pointer.0 = getelementptr [1024 x float]* @A, i64 0, i64 %indVar.0
%A.vector_pointer = bitcast float* %A.pointer.0 to <2 x float>*
%A.vector = load <2 x float>* %A.vector_pointer
%B.pointer.0 = getelementptr [1024 x float]* @B, i64 0, i64 0
%B.scalar.0 = load float* %B.pointer.0
%B.vector = shufflevector <1 x float> %B.scalar.0, <1 x float> undef,
                          <2 x i32> zeroinitializer
%result = add <2 x float> %A.vector, %B.vector
%C.pointer.0 = getelementptr [1024 x float]* @C, i64 0, i64 %indVar.0
%C.vector_pointer = bitcast float* %C.pointer.0 to <2 x float>*
store <2 x float> %result, <2 x float>* %C.vector_pointer
```

Listing 8: *Optimized vector code for listing 5: Usage of a full vector load for array A, a scalar load plus a splat for array B and a full vector store for array C*

In general a single vectorized loop has still a large load and store overhead and will probably not give huge performance improvements. Polyhedral optimizations that target vector code generation should therefore make sure that most memory accesses can be hoisted out of the new inner loops either directly or after some unrolling. This however is not part of the vector code generation, but just clever usage of existing compiler technology. Some examples how to generate good vector code are shown in Section 8.

Part III

EXPERIMENTS

8

MATRIX MULTIPLICATION - VECTORIZED

Matrix multiplication is a well known computation kernel, not just because of its simple structure, but especially because of its importance as a building block of many complex algorithms. To obtain good performance an optimized version of the matrix multiplication kernel is crucial. Such a kernel can be obtained by linking with special linear algebra libraries like BLAS [18] or their vendor optimized counterparts. Yet, a more convenient approach is to rely on the compiler to optimize such kernels automatically. By analyzing and optimizing a challenging variant of matrix multiplication we show what improvements are still possible over current compilers.

Listing 9 shows a plain matrix multiplication kernel that multiplies two 32x32 matrices and adds the obtained result to a third matrix. The kernel is not optimized in terms of vector intrinsics or any special loop structure, but a competitive compiler should create such a loop structure automatically. For the shown kernel this is not straightforward, as all memory accesses have unit stride in respect to different loop dimensions. If the loop nest is vectorized along one dimension only one memory access has unit stride, whereas the other two memory accesses will have non unit strides. With clever use of unrolling, loop interchange and strip mining, the cost of those non-unit stride accesses can be reduced, such that the use of SIMD instructions becomes profitable.

```
void matmul(float *restrict A, float *restrict B, float *restrict C) {
  for (i=0; i < 32; i++)
    for (j=0; j < 32; j++)
      for (k=0; k < 32; k++)
        C[i][j] += A[k][i] * B[j][k];
}
```

Listing 9: *Matrix multiplication kernel where each memory access has a unit stride in respect to another loop dimension*

Figure 23 shows the performance of this kernel with the different steps that can be taken to optimize code with Polly. The baseline is LLVM with all optimizations enabled, which is equivalent to the use of `clang -O3`. At version 2.8 LLVM does not change the loop structure of the kernel at all and no SIMD instructions are created. Even if LLVM does not apply neither vectorization nor memory access optimizations, good scalar code is created. GCC 4.4.5 does also not apply any loop transformations nor does it apply any vectorization. The performance of the GCC generated code shows that its scalar code is not as good as the code created by LLVM. In contrast ICC 11.1 performs loop transformations and introduces SIMD instructions. The code it generates is almost twice as fast as the one generated with LLVM. Yet, the assembly of the ICC generated code contains a large amount of scalar loads, which suggests that further optimization are possible.

Now Polly is used to generate optimized vector code. We lower the kernel to LLVM-IR and execute the Polly optimizations on it. After Polly's code generation the LLVM standard optimizations are scheduled to remove redundant scalar operations introduced by Polly. The first run shown is the Polly identity transformation which translates the kernel into the polyhedral model and recreates the LLVM-IR from the polyhedral representation without applying any transformations. This example shows, that the indirection through Polly does not add any notable overhead. The run time of the generated code does not increase.

```
for (k=0; k < 32; k++)
  for (j=0; j < 32; j+=4)
    for (i=0; i < 32; i++)
      for (jj=j; jj < j + 4; j++)
        C[i][jj] += A[k][i] * B[jj][k];
```

Listing 10: *Matrix multiplication kernel with loop structure prepared for vectorization*

The first transformation with Polly (+StripMine) changes the loop structure to improve data-locality and, more important, to expose a trivially vectorizable loop. We derive the structure of the new loop nest with the help of a research tool developed by Stock et al. [44]. The abstract description of the loop structure is translated into a polyhedral schedule and this schedule is manually imported into Polly. As can be seen in Listing 10 an innermost trivially vectorizable *jj* loop is generated. In addition the *i* loop is moved deeper into the loop structure, a transformation that later enables further optimizations. Even the simple change of the loop structure increases the performance by 19%.

```
for (k=0; k < 32; k++)
  for (j=0; j < 32; j+=4)
    for (i=0; i < 32; i++)
      C[i][j:j+3] += A[k][i] * B[j:3][k];
```

Listing 11: *Vectorized matrix multiplication kernel*

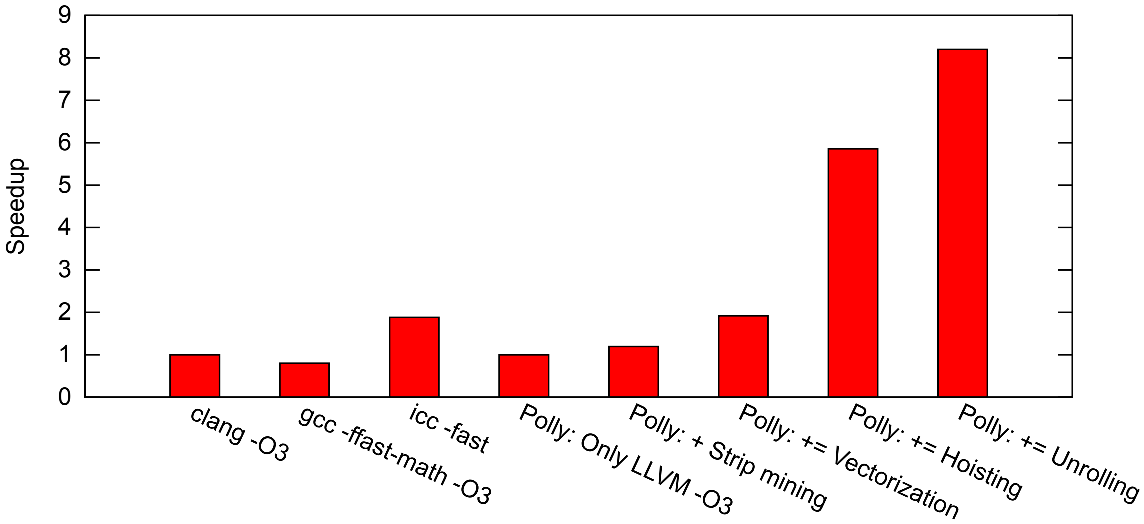


Figure 23: *10000 times 32x32 float matrix multiplication on Intel i5 CPU M 520 @ 2.40GHz (Options to Polly are cumulative, ICC 11.1, GCC 4.4.5, Clang 2.8)*

In the next run (+= Vectorization) we take advantage of the previously created trivially vectorizable loop. Instead of creating the innermost loop, we generate SIMD operations. As Polly can prove that the innermost loop does not carry any dependences, it can, if requested, apply the necessary transformations fully automatically. As described in Section 7.3.3, Polly automatically generates full vector loads for the access to C, a stride zero load for the access to A as well as scalar loads for the elements loaded from B. Since LLVM is able to prove aligned vector accesses, fast, aligned vector operations are generated by the LLVM X86 backend. At this point, the performance of the Polly optimized code is twice as good as the LLVM base line and the performance of ICC is reached.

In the innermost there is still an inefficient load from B. This load combines four elements distributed in memory and requires four scalar loads to initialize the SIMD vector. However, due to our previous changes to the loop structure the loads from B are invariant in the innermost loop. As Polly can prove that the i loop is always executed at least once, the loads from B can be hoisted out of this loop. The number for Polly (+= Hoisting) shows, that this transformation can triple the performance such that Polly easily outperforms ICC.

Finally, it is possible to trade code size for performance. By increasing the unrolling limits in the LLVM unrolling pass the inner two loops can be fully unrolled. As can be seen this leads to further increases in performance. The overall speedup we achieve is 8x compared to `clang -O3` and 4x over `icc -fast`.

We have shown that Polly significantly improves performance in comparison to ICC and Clang/LLVM by automatically generating optimized (SIMD) code. Only to calculate the optimal loop structure we use an external tool. We have seen that the largest performance improvements appear not because of the SIMD code introduced, but because of further optimization opportunities exposed through our preparing loop transformations. As subsequent transformations are already performed in Polly/LLVM automatically, the remaining problem is now the calculation of loop structure that exposes sufficient optimization opportunities. The polyhedral abstractions provide a powerful instrument to target this problem.

9

AUTOMATIC OPTIMIZATION OF THE POLYBENCH BENCHMARKS

9.1 THE IDENTITY TRANSFORMATION

To understand if the use of Polly has any negative impact on the quality of the generated code we analyze the overhead of the Polly identity transformation. The identity transformation translates from LLVM-IR to the polyhedral representation and back to LLVM-IR without applying any polyhedral transformations in between. It is run by the command `pollycc -fpolly` and the code it generates is compared to code compiled with plain `clang -O3`.

For our tests we use the benchmarks provided by the PolyBench[1] 2.0 test suite. They are used unmodified with a single exception. The type of some induction variables is changed from int to long to remove implicit type casts which would currently prevent the detection of some valid SCoPs. Except this change, the detection of SCoPs is fully automatic and does in particular not use any annotations in the source code. Furthermore, all tests are performed on an Intel® Xeon® X5670 system.

Looking at the results of the tests as presented in Figure 24, we can see that only for two benchmarks the run time changes significantly. For reg_detect we get a slowdown of 15%, because LLVM seems to generate some inefficient unrolling when simplifying the code generated by Polly. In the case of jacobe-1d-imper, there is an 18% speedup. However, this speedup is not caused by Polly, but is caused by some pass ordering issues

---

[1] http://www.cse.ohio-state.edu/p̃ouchet/software/polybench/

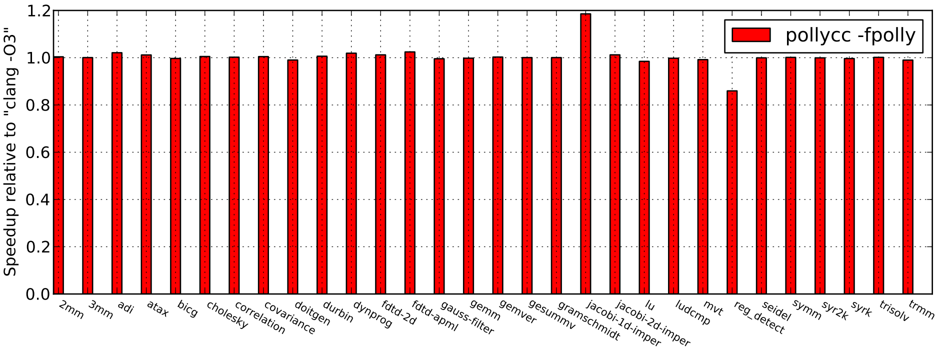
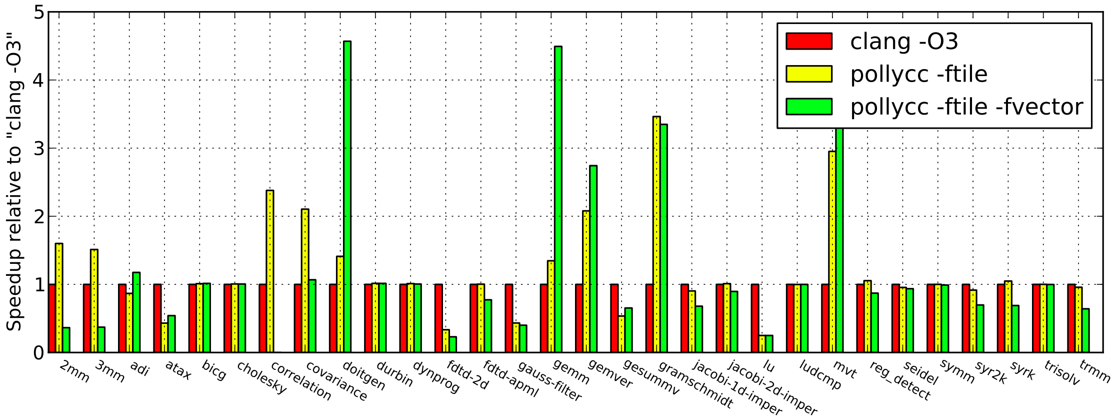
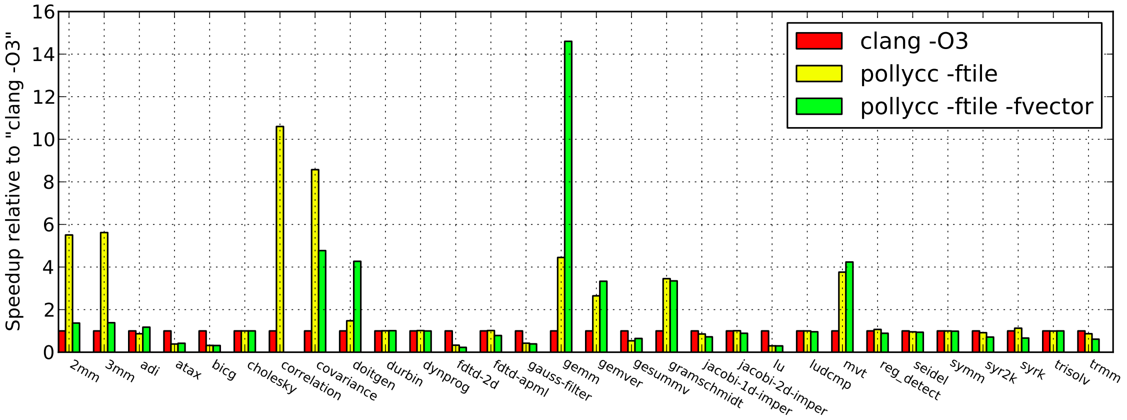


Figure 24: *Runtime overhead introduced by Polly*



(a) *Small data size*



(b) *Large data size*

Figure 25: *Sequential speedup of pollycc compared to clang (Both current from 25/03/2011, Benchmark: Polybench 2.0, run on Intel® Core™ Xeon X5670 CPU @ 2.93GHz)*

in LLVM. Those issues are hidden, if the LLVM standard optimization passes are run a second time to clean up the LLVM-IR produced by the Polly code generation. Except of these two cases, all other 28 benchmarks show performance changes of less than 2% and on average a run time increase of less than 0.4%. This shows clearly, that the use of Polly does in most cases not introduce significant overhead.

9.2 CREATING OPTIMIZED SEQUENTIAL CODE WITH PLUTO

To investigate the benefits fully automatic optimizations can currently provide for single thread performance we connect Polly with PoCC[2] and test the benefits of Pluto based tiling and vectorization. Even though Pluto based tiling is focusing on parallel performance, it can already give a first impression what polyhedral optimizations can achieve. Our tests were run again on the PolyBench 2.0 test suite on an Intel® Xeon® X5670 system. All tests are run with double precision floating point numbers on both small and larger data sizes.

---

[2] PoCC-rc3.1 as available from `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`

The test results presented in Figure 25a show 7 benchmarks, where tiling decreases the performance. Those are most of the time benchmarks, where tiling is not beneficial in general or where the problem size is too small to benefit from tiling. Due to the introduction of more complex control flow, the run time of those test cases increases. This is expected, as Polly currently does not include a heuristic to determine, if the use tiling will be profitable. For 9 of 30 benchmarks, Pluto based tiling can provide significant performance improvements even on small data sizes. For larger data sizes we obtain in average 2x speedup with 4 benchmarks showing more than 4x speedup and two even reaching 8x speedup. This is promising, as we have not yet analyzed the benchmarks intensively and the calculated schedules are most probably not yet optimal. Looking at the SIMD code generation we see that for two cases Pluto based vectorization can further increase the speedup. This is for example the case for gemm where additional use of vector operations leads to an overall speedup of 14x. To emphasize, this is still single thread performance and the speedup is only obtained by a more efficient use of caches and SIMD units. We also see that in many cases vectorization actually decreases the performance of the code. The reason here is that Pluto currently does not include any useful heuristics to decide how to vectorize, but always vectorizes the innermost parallel loop. In case this loop has no stride one accesses, the vectorization of this loop is often not beneficial. We believe, that by developing better heuristics for vectorization, benefits as seen for gemm can be shown for more benchmarks.

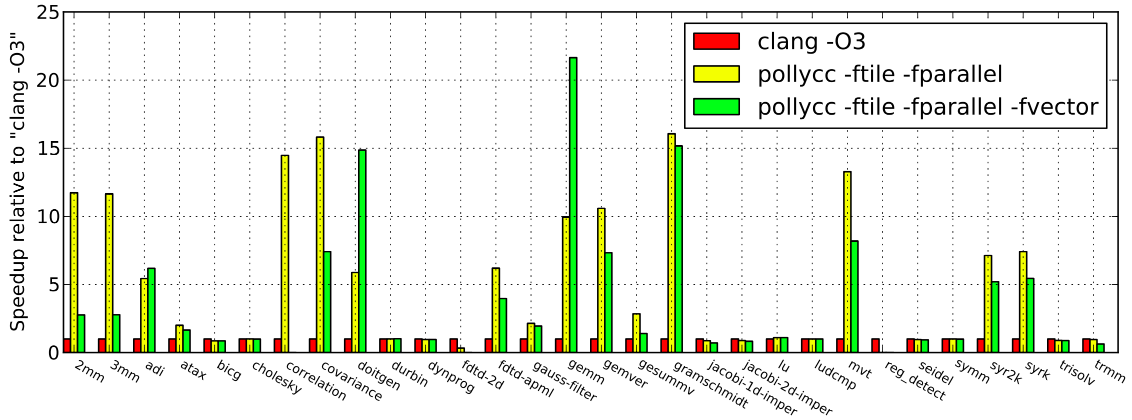
9.3 CREATING OPTIMIZED PARALLEL CODE WITH PLUTO

We also analyze the performance benefits that we can obtain by generating OpenMP parallel code. For this we start with tiling and vectorization. Then, we use Pluto's parallel mode to expose parallelism and we use Polly to create OpenMP code that takes advantage of this parallelism. All tests were run on a 2 socket, 12 core, 24 HT[3] threads Intel® Xeon® X5670 system.

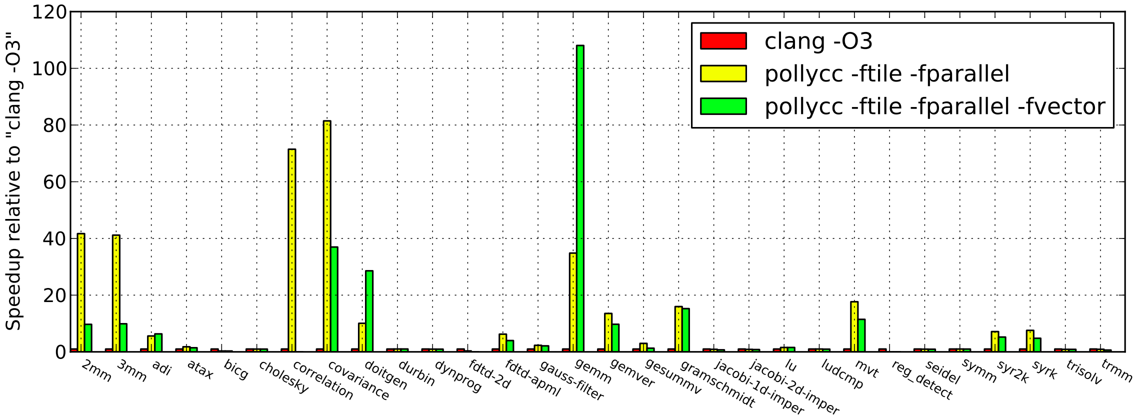
As can be seen in Figure 26, 16 of 30 benchmarks benefit from parallel execution. Even with small data sizes 8 benchmarks reach more than 10x speedup. For larger data sizes we obtain in average 12x speedup and there exist even a case that shows more than 80x speedup. We believe this already shows clearly that there exists a larger set of benchmarks that benefit from automatic parallelization as it can be performed in Polly. Together with vectorization, the gemm kernel is again the top performer with an overall speedup of more than 100x. As can be seen, parallelism combined with tiling and vectorization can show significant performance improvements for a large number of benchmarks. And again, we did not yet optimize the tests intensively such that we expect further improvements. The limited benefits vectorization currently provides suggest another area where further performance improvements may be possible.

---

[3] HyperThreading



(a) *Small data size*



(b) *Large data size*

Figure 26: *Parallel speedup of pollycc compared to clang (Both current from 25/03/2011, Benchmark: Polybench 2.0, run on Intel® Core™ Xeon X5670 CPU @ 2.93GHz)*

10

CONCLUSION

In this thesis we presented Polly, a framework to apply polyhedral transformations on a low level intermediate representation. We have shown how to automatically extract the static control parts of a program, how to translate them into a polyhedral description, how to apply transformations on this description and finally, how to regenerate optimized program code. The process described is programming language independent and does not require the input code to follow any syntactical form. Instead, we analyze a program semantically such that different textual incarnations of semantically equivalent constructs are supported transparently. As a result, code that contains constructs like goto-loops or pointer arithmetic can be optimized.

To represent the code Polly uses a modern polyhedral description based on an integer set library with support for a detailed data-flow analysis. This analysis is used to detect different forms of parallelism. For outermost parallel loops we introduce calls to an OpenMP runtime and in case of innermost parallel loops we create SIMD code. For the generated SIMD code we automatically determine the optimal vector instructions, such that polyhedral optimizers can focus on finding an execution order that exposes opportunities for generating effective vector code. Furthermore, we have defined a generic interface for external optimizers. It was used to connect PoCC,[1] a collection of polyhedral tools which includes the advanced data-locality and parallelism optimizer Pluto.

To understand the effectiveness of Polly, we optimized the Polybench 2.0 test suite and analyzed the results. The indirection through the polyhedral representation does not introduce any overhead in the generated code. In case the Pluto optimizations are used, we have seen significant speedups that reached 14x without introducing thread level parallelism and more than 100x when taking advantage of thread level parallelism.[2] In average we reached a 2x speedup without and a 12x speedup with thread level parallelism.[3] As the focus of this work was more on the infrastructure than on the optimizer, we believe these numbers can be further improved.

At this stage, we have finished the core of Polly, but to optimize code that commonly appears in larger applications further extensions are necessary. The main areas that need work are type cast operations, possible integer overflows and multi-dimensional variable sized arrays (multi-dimensional arrays with static size are already supported). Furthermore, we have seen that Pluto does not yet produce loop nests that allow effective SIMD code generation, such that currently only manually calculated optimizations show the benefits of our SIMD back end. Work to extend the Pluto algorithm to expose better SIMD code generation opportunities is planned. Another topic that we plan to target is the support for memory layout transformations.

---

[1] `http://pocc.sf.net`

[2] For the gemm kernel and large data size

[3] For large data sizes

Polly can apply advanced, polyhedral transformations without the need of any source code annotations or the use of source-to-source techniques. As it works directly inside a compiler, it can be used to optimize existing programs fully automatically. We have already shown significant performance improvements on a number of computation kernels with the help of the Pluto optimizers and believe that the Polly infrastructure will simplify the development of new, advanced optimizations. As it abstracts away all language specific details and code generation problems, it allows to focus on the high-level optimization problems. With the interface for external optimizers new transformations can be added to Polly without the need to understand all LLVM internals. Open research topics that may benefit from Polly are for example polyhedral optimizations in just-in-time environments, automatic offloading of calculations to vector accelerators or polyhedral optimizations in high level synthesis. As LLVM provides a good infrastructure for these topics,[4] even solutions that target several of these problems at once could be investigated.

---

[4] In LLVM provides a JIT compiler as well as PTX and Verilog code generation.

Part IV

APPENDIX

LIST OF FIGURES

Figure 1 Static compilation using the llvm toolchain 8
Figure 2 CFG of the example program 14
Figure 3 The dominator tree for the CFG in Figure 2 15
Figure 4 A simple Region 16
Figure 5 Transform a refined region to a simple region 17
Figure 6 Simple (solid border) and refined (dashed border) regions in a CFG. 17
Figure 7 The elements of a scalar evolution 19
Figure 8 A loop nest and the two basic sets used to describe the valid loop iterations. 24
Figure 9 Architecture of Polly 30
Figure 10 Passes available in Polly as printed by 'opt -load LLVMPolly.so -help' 31
Figure 11 pollycc command line options 32
Figure 12 A valid SCoP (validity defined on ASTs) 34
Figure 13 Two valid SCoPs (validity defined based on semantics) and their canonical counterparts 35
Figure 14 An example of a SCoP and its polyhedral representation. 38
Figure 15 Translation from C over LLVM-IR to polyhedral shown on a simple example 41
Figure 16 Region based SCoP detection 42
Figure 17 Simple SCoP before independent block pass 45
Figure 18 Simple SCoP after independent block pass 46
Figure 19 JSCoP file of a matrix multiplication kernel 53
Figure 20 MayWrites and accesses to memory ranges 56
Figure 21 Three kinds of vector loads: v1 is a stride-one vector load, v2 is a stride-zero vector load and v3 is a complex vector load 65
Figure 22 Two kind of vector stores: v1 is a stride-one vector store and v3 is a complex vector store 65
Figure 23 10000 times 32x32 float matrix multiplication on Intel i5 CPU M 520 @ 2.40GHz (Options to Polly are cumulative, ICC 11.1, GCC 4.4.5, Clang 2.8) 73
Figure 24 Runtime overhead introduced by Polly 75
Figure 25 Sequential speedup of pollycc compared to clang (Both current from 25/03/2011, Benchmark: Polybench 2.0, run on Intel® Core™ Xeon X5670 CPU @ 2.93GHz) 76

Figure 26 Parallel speedup of pollycc compared to clang (Both current from 25/03/2011, Benchmark: Polybench 2.0, run on Intel® Core™ Xeon X5670 CPU @ 2.93GHz) 78

LIST OF TABLES

LIST OF LISTINGS

1 Two different ASTs generated from the same SCoP. 58
2 A single loop nest hides parallelism 60
3 Two separate loop nests expose parallelism 60
4 Two steps to vectorize a loop 64
5 Example of a trivially vectorizable loop 64
6 The LLVM-IR of the loop body in listing 5 65
7 Unoptimized vector code for listing 5 66
8 Optimized vector code for listing 5: Usage of a full vector load for array A, a scalar load plus a splat for array B and a full vector store for array C 67
9 Matrix multiplication kernel where each memory access has a unit stride in respect to another loop dimension 71
10 Matrix multiplication kernel with loop structure prepared for vectorization 72
11 Vectorized matrix multiplication kernel 72

LIST OF ACRONYMS

VMaIR	
AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subprogram
CFG	Control Flow Graph
CLOOG	Chunky Loop Generator
CISC	Complex instruction set computing
GCC	GNU Compiler Collection
GHC	Glasgow Haskell Compiler
GPGPU	General-purpose computing on graphics processing units
HLS	High Level Synthesis
ISL	integer set library
JSCOP	JSON SCoP
JSON	JavaScript Object Notation
LLVM	Low Level Virtual Machine
LLVM-IR	LLVM Intermediate Representation
OPENMP	Open Multi-Processing
POCC	Polyhedral Compilation Collection
SCOP	Static Control Part
SSA	Static Single Assignment
SIMD	Single Instruction Multiple Data

REFERENCES

[1] DragonEgg. http://dragonegg.llvm.org/, 2010.

[2] Icedtea. http://icedtea.classpath.org/, 2010.

[3] LLVM Lua. http://code.google.com/p/llvm-lua/, 2010.

[4] LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html, 2011.

[5] LLVM Ruby. http://www.llvmruby.org/, 2010.

[6] Unladen Swallow. http://code.google.com/p/unladen-swallow/, 2010.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, Aug. 2006.

[8] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of Recurrences - A Method to Expedite the Evaluation of Closed-form Functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, pages 242–249. ACM, 1994.

[9] S. Baghdadi, A. Größlinger, and A. Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, jul 2010.

[10] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin / Heidelberg, 2010.

[11] C. Bastoul. Code Generation in the Polyhedral Model is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Sept. 2004.

[12] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. . In *ETAPS International Conference on Compiler Construction (CC'2010)*, pages 283–303. Springer Verlag, Mar. 2010.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference On Programming Language Design and Implementation*, PLDI '08, pages 101–113. ACM, 2008.

[14] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, 2011.

[15] C. Chen, J. Chame, and M. Hall. CHiLL: A Framework for Composing High-Level Loop Transformations. Technical Report 08-897, June 2008.

[16] P. Clauss and B. Meister. Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests. *SIGARCH Computer Architecture News*, 28:11–19, Mar. 2000.

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

[18] J. Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard. *International Journal of High Performance Computing Applications*, 16:1, February 2002.

[19] R. A. V. Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *In Proceedings of the International Conference on Compiler Construction*, ETAPS CC 'O1, pages 118–132, 2001.

[20] P. Feautrier. Dataflow analysis of array and scalar references. 20:23–53, 1991.

[21] P. Feautrier. Automatic Parallelization in the Polytope Model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*, pages 79–103. Springer Berlin / Heidelberg, 1996.

[22] Free Software Foundation Inc. Welcome to the Home of GOMP. `http://gcc.gnu.org/projects/gomp/`, 2010.

[23] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic Composition of Loop Transformations For Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006.

[24] M. Griebl and C. Lengauer. The Loop Parallelizer LooPo. In *Languages and Compilers for Parallel Computing*, volume 1239 of *Lecture Notes in Computer Science*, pages 603–604. Springer Berlin / Heidelberg, 1997.

[25] M. Griebl, P. Feautrier, and C. Lengauer. Index Set Splitting. *International Journal of Parallel Programming*, 28:607–631, 2000.

[26] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 237–248, 2007.

[27] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185. ACM, 1994.

[28] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Hyperplane Based Approach for Optimizing Spatial Locality in Loop Nests. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 69–76. ACM, 1998.

[29] W. Kelly and W. Pugh. A Unifying Framework for Iteration Reordering Transformations. In *Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP.*, volume 1, pages 153 –162 vol.1, Apr. 1995.

[30] C. Lattner. LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[31] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[32] M. Levoy. Experimental platforms for computational photography. *IEEE Computer Graphics and Applications*, 30:81–87, 2010.

[33] LLVM Project. Developer Meeting 2010. `http://llvm.org/devmtg/2010-11/`, 2010.

[34] LLVM Project. Users of LLVM. `http://llvm.org/Users.html`, 2010.

[35] TIOBE Software BV. TIOBE Programming Community Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, Jan. 2011.

[36] M. Perache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Euro-Par 2008 - Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin / Heidelberg, 2008.

[37] S. Pop. Analysis of Induction Variables Using Chains of Recurrences: Extensions. Master's thesis, Universite Louis Pasteur, Strasbourg, 2003.

[38] R. T. Prosser. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

[39] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 546–566. 1994.

[40] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan. CHiMPS: A High-level Compilation Flow for Hybrid CPU-FPGA Architectures. In *Proceedings of the 16th International ACM/SIGDA symposium on Field Programmable Gate Arrays*, FPGA '08, pages 261–261. ACM, 2008.

[41] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of Efficient Nested Loops from Polyhedra. *International Journal on Parallel Programming*, 28:469–498, October 2000.

[42] S. Rajopadhye, L. Renghanarayana, G. Gupta, and M. M. Strout. *Computations on Iteration Spaces*, chapter 15. CRC Press, second edition, 2008.

[43] T. Risset, S. Derrien, P. Quinton, and S. Rajopadhye. High-Level Synthesis of Loops Using the Polyhedral Model. In *High-Level Synthesis : From Algorithm to Digital Circuit*, pages 215–230. 2008.

[44] K. Stock, T. Henretty, I. Murugandi, and P. Sadayappan. Model-Driven SIMD Code Generation for a Multi-Resolution Tensor Kernel. In *IPDPS*, 2011. Accepted for publication.

[45] L. Tarrataca, A. C. Santos, and J. a. M. P. Cardoso. The Current Feasibility of Gesture Recognition for a Smartphone Using J2ME. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1642–1649. ACM, 2009.

[46] D. A. Terei and M. M. Chakravarty. An LLVM Backend for GHC. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 109–120. ACM, 2010.

[47] The mono project. Mono LLVM. `http://icedtea.classpath.org/`, 2010.

[48] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, jan 2010. URL `http://hal.inria.fr/inria-00551516/en/`.

[49] J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. In *BPM '08: Proceedings of the 6th International Conference on Business Process Management*, pages 100–115. Springer-Verlag, 2008.

[50] S. Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.

[51] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127 –134, May 2010.

[52] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-Time Detection and Tracking for Augmented Reality on Mobile Phones. *IEEE Transactions on Visualization and Computer Graphics*, 16:355–368, 2010.

[53] Y.-C. Wang, S. Pang, and K.-T. Cheng. A GPU-accelerated Face Annotation System for Smartphones. In *Proceedings of the International Conference on Multimedia*, MM '10, pages 1667–1668. ACM, 2010.

[54] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. wei Liao, C. wen Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29:31–37, 1994.

[55] Y. Xiong and K. Pulli. Fast Panorama Stitching for High-quality Panoramic Images on Mobile Phones. *Consumer Electronics, IEEE Transactions on*, 56:298 –306, May 2010.

[56] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Security and Privacy, IEEE Symposium on*, 0:79–93, 2009.

[57] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level Optimization for High-level Synthesis and FPGA-based Acceleration. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, 2010.

[58] E. V. Zima. On Computational Properties of Chains of Recurrences. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, ISSAC '01, pages 345–. ACM, 2001.

DECLARATION

Hiermit versichere ich, dass ich diese Diplomarbeit selbsständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, Germany, April 2011

Tobias Christian Grosser