Parallel Processing Letters

POLLY—PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION

TOBIAS GROSSER
Parkas Group, Computer Science Department, Ècole Normale Supèrieure / INRIA
45 Rue d'Ulm, Paris, 75005, France

ARMIN GROESSLINGER and CHRISTIAN LENGAUER
Programming Group, Department of Informatics and Mathematics, University of Passau
Innstraße 33, Passau, 94032, Germany

Received February 2012
Revised May 2012
Communicated by S. G. Akl

ABSTRACT

The polyhedral model for loop parallelization has proved to be an effective tool for advanced optimization and automatic parallelization of programs in higher-level languages. Yet, to integrate such optimizations seamlessly into production compilers, they must be performed on the compiler's internal, low-level, intermediate representation (IR). With Polly, we present an infrastructure for polyhedral optimizations on such an IR. We describe the detection of program parts amenable to a polyhedral optimization (so-called static control parts), their translation to a Z-polyhedral representation, optimizations on this representation and the generation of optimized IR code. Furthermore, we define an interface for connecting external optimizers and present a novel way of using the parallelism they introduce to generate SIMD and OpenMP code. To evaluate Polly, we compile the PolyBench 2.0 benchmarks fully automatically with PLuTo as external optimizer and parallelizer. We can report on significant speedups.

Keywords: Automatic loop optimization, LLVM, OpenMP, polyhedron model, SIMD, tiling

1. Introduction

1.1. *Motivation*

Modern hardware provides many possibilities of executing a program efficiently. Multiple cache levels help to exploit data locality, and there are various ways of taking advantage of parallelism. Short vector instruction sets, such as provided by Intel SSE and AVX, IBM AltiVec or ARM NEON, offer fine-grained parallelism. Dedicated vector accelerators or GPUs supporting general-purpose computing offer massive parallelism. On platforms such as the IBM Cell, Intel SandyBridge or AMD Fusion similar accelerators are available, tightly integrated with general-purpose

CPUs. Finally, an increasing number of cores exists even on ultra-mobile platforms. The effective use of all available resources is essential for highest efficiency. Consequently, well optimized programs are required.

Traditionally, such optimizations are performed by translating performance-critical parts of a software system to a lower-level language, e.g., C or C++, and optimizing them manually. This is a difficult task: 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 compounded if hints for the compiler are needed, such as the annotation of parallel loops or of code to be delegated to an accelerator. As a result, domain experts are required for specifying such optimizations.

Even if they succeed, other problems remain. For one, such optimizations are extremely platform-dependent and often not even portable between different micro-architectures. Consequently, programs need to be optimized for every target architecture individually. This is a complex undertaking. Today, an application may target at the same time ARM-based smartphones, Intel Atom-based 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 impossible from the outset. High-level language features, such as the iterators and the FOREACH loop of C++, hinder manual loop optimizations, since they require manual inlining. Languages such as Java, Python and JavaScript provide no support for portable low-level optimizations. Even programs compiled for the Google Native Client [29], 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 or even impossible.

Fortunately, powerful algorithms are available for optimizing computation-intensive programs automatically. Wilson et al. [28] implemented an automatic parallelization and data locality optimizations based on unimodular transformations in the SUIF compiler, Feautrier [10] developed an algorithm for calculating a parallel execution order from scratch and Griebl et al. [12] developed LooPo, a complete infrastructure for the comparative study of polyhedral algorithms and concepts. Furthermore, Bondhugula et al. [6] created PLuTo, an advanced data locality optimizer that exposes thread and SIMD parallelism simultaneously. There are also methods for offloading calculations to accelerators [2] and even techniques for synthesizing high-performance hardware [22]. All these techniques are part of a large set of advanced optimization algorithms based on polyhedral concepts.

However, the use of these advanced algorithms is currently limited. Most of them are implemented in source-to-source compilers with language-specific front ends for the extraction of relevant code regions. This often requires the source code of these regions to be in a canonical form and to be void of 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, since 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. This limited integration prevents an effective use of polyhedral tools for compiler-internal optimizations. As a result, influencing performance-related decisions of the compiler is difficult and the resulting target programs often suffer 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 contemporary 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 lack integration in a production compiler and thus, have no significant impact.

1.2. *Contributions*

We introduce Polly,[a] a tool for the automatic polyhedral optimization of the low-level IR representation of real programs. We present the entire infrastructure of Polly. Our treatise is based on the diploma thesis of the first author [13]; some selected aspects have been introduced earlier [14].

Polly detects and extracts relevant code regions without any human interaction. Since Polly works on LLVM's intermediate representation (LLVM-IR), it is independent of the programming language used and supports constructs such as C++ iterators, pointer arithmetic and goto-based loops transparently. It is based on an advanced polyhedral library [27] that can model integer wrapping and that provides a state-of-the-art dependence analysis. Due to a simple file interface, it is possible to apply transformations manually or to use an external optimizer. We have used this interface to integrate PLuTo, a modern data locality optimizer and parallelizer. Polly automatically detects existing and newly exposed parallelism and can take advantage of it through the integrated SIMD and OpenMP code generation. This allows automatic optimizers to focus on the parallelization problem and to offload low level SIMD and OpenMP code generation to Polly.

The remainder is organized as follows. After discussing related work in Section 2, Section 3 presents the general architecture of Polly. Section 4 describes the detection of interesting code regions and their translation into a polyhedral description. Section 5 addresses the analysis and transformation of this description and proposes an exchange format for external optimizers. Section 6 describes how we regenerate LLVM-IR and, subsequently, OpenMP and SIMD code. Section 7 reports on some experiments and evaluation. Finally, Section 9 concludes.

[a]The name "Polly" is a combination of **Poly**hedral and **LL**VM.

2. Related Work

Automatic optimization using the polyhedron model has been studied and implemented in several frameworks. Most systems to date perform a source-to-source transformation, e.g., PIPS [15], LooPo [12], PLuTo [6], PoCC,[b] AlphaZ[c] and the commercial R-Stream source-to-source compiler [17]. More recently, polyhedral transformations have also been implemented at the level of the intermediate representation to overcome the restiction to subsets of "real" languages (cf. Section 1). In the Wrap-IT [11] research project (based on Open64) and the IBM XL compiler [5], polyhedral optimizations act on the compiler's high-level intermediate representation.

GRAPHITE [24] was among the first to demonstrate the integration of polyhedral tools into a production compiler. It is independent of the source language, as it extracts polyhedral information from GIMPLE, GCC's low-level intermediate representation. On this polyhedral information, GRAPHITE performs a set of classical loop optimizations. Even though GRAPHITE has demonstrated the feasibility of this approach, it still has several important shortcomings. Relevant code regions are detected in an unstructured way, which limits the size of the code that can be optimized. The polyhedral representation is based on rational polyhedra which requires conservative approximations and, in addition, makes it impossible to analyze code that contains modulo arithmetic or cast expressions. Also, GRAPHITE does not provide advanced polyhedral optimizations and has not been connected to an external optimizer, yet. Finally, GRAPHITE requires memory accesses to be expressed as array accesses and cannot analyze code that contains pointer accesses.

Polly offers several innovations over previous approaches. Working on a compiler IR at a level even lower than the one used by GRAPHITE, it is able to offer solutions for many of the open problems in GRAPHITE. It replaces ad-hoc SCoP detection approaches with a structured approach based on control flow regions. Polyhedral calculations are consistently performed on integer sets, instead of falling back to rational polyhedra. An interface for external optimizers is provided and we show that the PLuTo algorithm can be applied to a low-level IR. Polly also removes the requirement of explicit array accesses and makes memory accesses based on pointer arithmetic first-level constructs.

3. Polly

3.1. *Architecture*

Polly is a framework that uses polyhedral techniques to optimize an LLVM-IR program for data locality and parallelism. Similarly to GRAPHITE, it takes a three-step approach. First it detects the parts of a program that will be optimized and translates them to a polyhedral representation, then it analyzes and optimizes the

[b] `http://pocc.sf.net`
[c] `http://www.cs.colostate.edu/AlphaZ/`

polyhedral representation, and finally it regenerates optimized program code. The three steps are implemented by a set of LLVM-IR passes grouped into front end, middle part and back end. The overall architecture is depicted in Figure 1.

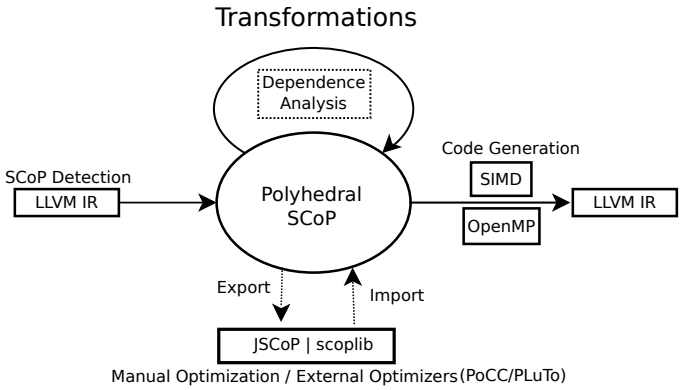


Fig. 1. Architecture of Polly

In the front end, the static control parts (SCoPs) of an input program are detected. SCoPs are the parts of a program that the front end analyzes and subsequently translates to a polyhedral representation. To keep Polly simple, we detect only SCoPs that match a certain canonical form. Code that is not in this form is canonicalized beforehand.

The middle part provides a polyhedral data flow analysis and is the place at which optimizations on the polyhedral representation are performed. There are two ways to perform optimizations. Either they are directly integrated into Polly, or the possible export and reimport of the polyhedral representation is used to apply manual optimizations or to connect an external optimizer. Polly includes direct support for PoCC (with the PLuTo optimizer) as an external optimizer. For internal optimizations, work has started to add an enhanced implementation of the PLuTo algorithm. The experiments in Section 7.2.2 and 7.2.3 have been performed with the well established PoCC optimizer.

In the back end, the original LLVM-IR is replaced by new code that is generated from the, possibly transformed, polyhedral representation. During code generation, we detect parallel loops and implement them as OpenMP parallel loops or SIMD instructions.

3.2. *How to Use Polly*

There are two ways to use Polly. The first is to invoke explicitly selected analyses and optimizations directly on an LLVM-IR file, the other is to use Polly as a compiler-internal optimizer.

The LLVM passes offered by Polly can be run individually. To this end, LLVM provides the tool `opt`, which can run an arbitrary list of passes on an LLVM-IR file. Thus, it is possible to use only parts of Polly, e.g., the SCoP detection or the dependence analysis, or to schedule selected optimizations.

A more comfortable way of using Polly is as an integrated part of a compiler. Polly can be loaded automatically with clang and gcc.[d] If loaded, Polly is run automatically at -O3. It schedules a predefined sequence of Polly passes, which applies polyhedral optimizations during normal compilation.

4. From LLVM-IR to a Polyhedral Description

LLVM-IR has been designed to perform low-level optimizations. For high-level optimizations, a more abstract representation is often better suited. The scalar evolution analysis, for example, abstracts from the instructions that calculate a certain scalar value and derives an abstract description of the result. This has been proven to be useful for scalar optimizations. For memory access and loop optimizations, Kelly and Pugh [18] proposed an abstraction based on integer polyhedra that is nowadays used in optimizers such as CHiLL [7], LooPo and PLuTo.

Polly uses polyhedral abstractions, but calculates them in an unusual way. Most optimizers derive a polyhedral description from a high-level programming language (cf. Section 2). In contrast, Polly analyzes a low-level intermediate representation.

This section describes our polyhedral representation. It explains which parts of a program can be analyzed, how these parts are detected, how their polyhedral description is derived, and which transformations are run to prepare the code for the analysis by Polly.

4.1. *What can be Translated?*

Polly optimizes the static control parts (SCoPs) of a function. The control flow and memory accesses of a SCoP are known at compile time. They can be described in detail and allow a precise analysis. Extensions for non-static control are due to Benabderrahmane et al. [4] and can, if required, be implemented in Polly.

SCoPs are usually defined syntactically on a high-level abstract syntax tree (AST). A common syntactic definition is as follows. A part of a program is a SCoP if the only control flow structures it contains are FOR loops and IF statements. For each loop, there exists a single integer induction variable that is incremented from a lower to an upper bound by a constant stride. Lower and upper bounds are expressions which are affine in parameters and surrounding loop induction variables; a parameter is any integer variable that is not modified inside the SCoP. IF conditions compare the value of two affine expressions. The only valid statements are assignments of the result of an expression to an array element. The expression

[d]See the DragonEgg project at `http://dragonegg.llvm.org`.

itself consists of side effect-free operators or function calls[e] with induction variables, parameters or array elements as operands. Array subscripts are affine expressions in induction variables and parameters. An example of a SCoP satisfying these syntactic criteria is given in Listing 1.

```
for (i = 0; i <= N; i++) {
    if (i <= N - 50)
S1:     A[5*i] = 1;
    else
S2:     A[3*i] = 2;

    for (j = 0; j <= N; j++)
S3:     B[i][2*j] = 3;
}
```

Listing 1: A static control part (SCoP)

SCoP 1

```
i = 0;

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

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

SCoP 1 as FOR loop

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

SCoP 2

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

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

SCoP 2 as a FOR loop

```
int A[1024];

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

Listing 2: Two SCoPs recognized semantically and their counterparts as FOR loops

By design, Polly does not work on the AST of a particular source language; instead, it optimizes a low-level representation that does not contain any high-level constructs. Hence, it must derive the high-level information. Polly recognizes a SCoP if it can establish that the low-level program part is semantically equivalent to a

[e]In LLVM IR, most operators are free of side effects. Information about the side effects of function calls and intrinsics is obtained via annotations, such as `pure`, or is deduced during existing LLVM analysis passes.

(syntactically defined) SCoP written in a high-level language. Thus, Polly is not restricted to specific programming language constructs. Listing 2 shows two SCoPs that are detected by Polly. To illustrate their semantics, we also show semantically equivalent counterparts written as FOR loops.

4.2. *A SCoP Defined on LLVM-IR*

In LLVM-IR, a SCoP is a subgraph of the control flow graph (CFG) that forms a single-entry-single-exit region and that is semantically equivalent to a classical SCoP. Verifying this is mostly straightforward. Only the derivation of affine functions and the possible aliasing of memory accesses is more involved and is consequently explained further.

To derive affine expressions in conditions, loop bounds and memory accesses, we take advantage of an LLVM analysis that describes them as scalar evolution (scev) [25]; scevs are conceptually more expressive than affine expressions. Thus, we must verify the equivalence of a scev with an affine expression. It is given if the scev consists only of integer constants, parameters, additions, multiplications with constants, maximum expressions or add recurrences that have constant steps. At present, we do not allow minimum, zero extend, sign extend or truncate expressions even though they can be conceptually represented in the polyhedral model.

Load and store instructions are the only instructions that can access memory. To derive an array access (with base address and subscript expression) from low-level pointer arithmetic and dereferences we need to be able to extract the base pointer from the scev that describes the memory location accessed. To model exact data access functions, we need to ensure that the scev describing the address offset is affine. If the access function is not affine, Polly can still represent the memory access, but needs to take conservative assumptions. Distinct base addresses in a SCoP must reference distinct memory spaces. In this context, a memory space 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 alias analysis technology which gives us exactly this information. If two base addresses are recognized as *must alias*, they are identical and modeled as the same array. If recognized as *no alias*, one of the two base addresses cannot yield an address derived from the other and they are modeled as different arrays. If recognized as *may alias*, LLVM is not certain and the SCoP is not accepted for further processing by Polly.

4.3. *The Polyhedral Representation*

To represent a SCoP, Polly uses a polyhedral description that is based on integer sets and maps (unions of Z-polyhedra representing a set of points or a relation between two sets of points) provided by the isl library [27].

A *SCoP* is a pair $(context, [statements])$ of a context and a list of statements. The context is an integer set that describes constraints on the parameters of the

SCoP, e.g., the fact that they are always positive or that the parameters are in certain relations. A Polly *statement* is a quadruple (*name*, *domain*, *schedule*, [*accesses*]) consisting of a name (e.g., *S1*), a domain, a schedule and a list of accesses. It represents a basic block in the SCoP and is the smallest unit that can be scheduled independently. The *domain* $\mathcal{D}$ of the statement is a named integer set that describes the set of different loop iterations in which the statement is executed. Its name corresponds to the name of the statement. The number of parameter dimensions matches the number of parameters in the SCoP. The number of set dimensions is equal to the number of loops that contain the statement and that are contained in the SCoP. Each set dimension is associated with one loop induction variable, the first dimension with the outermost loop. An iteration vector is a single element of the domain. Together with a statement it defines a *statement instance*.

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

$$\begin{aligned} \mathcal{D}_{S1} &= \{S1[i] : i \geq 0 \wedge i \leq N \wedge i \leq N - 50\} & \mathcal{D}_{S2} &= \{S2[i] : i \geq 0 \wedge i \leq N \wedge i > N - 50\} \\ \mathcal{S}_{S1} &= \{S1[i] \rightarrow [0, i, 0, 0, 0]\} & \mathcal{S}_{S2} &= \{S2[i] \rightarrow [0, i, 1, 0, 0]\} \\ \mathcal{A}_{S1} &= \{S1[i] \rightarrow A[5i]\} & \mathcal{A}_{S2} &= \{S2[i] \rightarrow A[3i]\} \end{aligned}$$

$$\begin{aligned} \mathcal{D}_{S3} &= \{S3[i, j] : i \geq 0 \wedge i \leq N \wedge j \geq 0 \wedge j \leq N\} \\ \mathcal{S}_{S3} &= \{S3[i, j] \rightarrow [0, i, 2, j, 0]\} \\ \mathcal{A}_{S3} &= \{S3[i, j] \rightarrow B[i][2j]\} \end{aligned}$$

Fig. 2. Polyhedral representation of the SCoP in Listing 1

The *schedule* $\mathcal{S}$ of a statement is an integer map that assigns to each iteration vector a multi-dimensional point in time. This defines the execution order of different statement instances in the final target code. Statement instance $Stmt_A[i]$ is executed before statement instance $Stmt_B[i']$ if $\mathcal{S}_{Stmt_A}(i)$ is lexicographically smaller than $\mathcal{S}_{Stmt_B}(i')$. It is valid to assign no execution time to an iteration vector. In this case, the corresponding statement instance is not executed. The assignment of multiple execution times to one iteration vector is presently not supported.

An access (*kind*, *relation*) is given by the kind of the access and the access relation. There are three kinds: read, write and may-write. The access relation $\mathcal{A}$ is an integer map that maps from the domain of a statement to 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. Unknown access functions are represented as may-write or read accesses to the entire array.

```
                    for (i = 0; i < 100; i++)
                    S4:     K[2*i] = 3;

bb:
  br label %bb1

bb1: ; Basic block of S4
  %indvar = phi i64 [ 0, %bb ], [ %indvar.next, %bb1 ]
; %indvar -> {0,+,1}<%bb1>
  %tmp = mul i64 %indvar, 2
  %gep = getelementptr [100 x float]* @K, i64 0, i64 %tmp
; %gep -> {@K,+,(2 * sizeof(float))}<%bb1>
  store float 3.000000e+00, float* %gep, 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
; Loop %bb1: backedge-taken count is 99
```

$$\mathcal{D}_{S4} = \{S4[i] : 0 \leq i < 100\}$$
$$\mathcal{S}_{S4} = \{S4[i] \rightarrow [0, i, 0]\}$$
$$\mathcal{A}_{S4} = \{S4[i] \rightarrow K[2i]\}$$

Fig. 3. Translation from C over LLVM-IR to polyhedral shown on a simple example

Figure 2 shows the polyhedral representation of the SCoP in Figure 1. The domain of *S1* is one-dimensional, since 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, since *S1* is also part of a conditional branch. The same holds for *S2*, but since *S2* is part of the ELSE branch of the condition, the negated constraint $i > N - 50$ is added to its domain. *S3* 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 to a five-dimensional timescale, which ensures the same execution order as in the original code. Furthermore, with $\mathcal{A}_{S1}$, $\mathcal{A}_{S2}$ and $\mathcal{A}_{S3}$, 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.

The same information can be derived from LLVM-IR. In Figure 3, we demonstrate the stepwise translation from C via LLVM-IR to our polyhedral representation. The C code is first lowered to LLVM-IR instructions which, among other things, calculate loop bounds, induction variables and array access functions. For instance, the address of the memory accessed by *S4* is calculated and the result is stored in `%gep`. To derive the access function from `%gep`, we calculate the scalar evolution expression $\{$`@K`$, +, (2 \cdot$ `sizeof(float)`$)\}$`<%bb1>`, which describes the mem-

ory address accessed. This expression is then divided into a base element K and a single-dimensional access function $2 \cdot i$, where i is a virtual induction variable counting the number of loop iterations. The access relation is $\{S4[i] \rightarrow K[2i]\}$. To calculate the domain of the statement, we call the scalar evolution analysis and ask for the number of times which the loop back edge is executed. In this example, the resulting expression is the integer constant 99. To derive the number of loop iterations, we simply add 1. With the knowledge that canonical induction variables always start at zero, we obtain the domain $\{S4[i] : 0 \leq i < 100\}$.

4.4. *Preparing Transformations*

Polly uses preparing transformations to increase the amount of detectable code. To keep the Polly front end simple, these transformations also convert the code to a canonical form. Canonicalization is performed via a set of transformations already available in LLVM and, additionally, some transformations especially developed for Polly.

4.4.1. *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`. Among others, 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 and induction variable canonicalization. Furthermore, we use a pass to transform each SCoP such that its CFG has a single entry edge and a single exit edge. Most passes are conceptually not necessary, but they simplify the implementation of Polly significantly.

4.4.2. *Independent blocks*

The independent block pass in Polly removes unnecessary dependences between basic blocks and creates basic blocks that can be scheduled freely. Furthermore, the independent block pass promotes scalar dependences, that cannot be removed, to accesses to single-element arrays. This ensures that the only scalar dependences that remain are references to induction variables or parameters. In particular, all ϕ-nodes, except those implementing loop induction variables, are promoted to memory operations. Since induction variables and parameters are replaced during code generation, Polly does not need to pay special attention to scalar dependences.

Unnecessary dependences are introduced regularly due to calculations of array indices, branch conditions or loop bounds which are often spread across several basic blocks. To remove such dependences, we duplicate all trivial scalar operations in each basic block. Thus, if an operation does not access memory and does not have any side effects, its results are not transferred from another basic block but are entirely recalculated in each basic block. Exceptions are parameters and induction variables,

which are not touched at all. The recalculation of scalar values introduces a notable amount of redundant code. In Section 7.2, we show that the normal LLVM cleanup passes can remove these redundant calculations entirely.

5. Polyhedral Analysis and Transformations

Polly performs its major analyses and transformations on the abstract polyhedral representation. We explain how we calculate data dependences, how we apply polyhedral transformations and how external optimizers interact with Polly.

5.1. *Dependence Analysis*

Polly provides an advanced dependence analysis implemented on top of the isl data flow analysis which uses techniques developed by Feautrier [9] and has also been influenced by Pugh [20]. It can model integer wrapping due to support for modulo constraints, it can be restricted to non-transitive dependences, and it allows access relations (not only functions) as well as may-write accesses. We believe that the latter two features do not exist in most existing data dependence implementations.

At present, Polly uses the dependence analysis to calculate read-after-write (flow), write-after-write (output) and write-after-read (anti) dependences 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 of the memory before the execution of the SCoP.

5.2. *Polyhedral Transformations*

There are two major ways of optimizing SCoPs: changing the execution order of the statement instances and changing the memory locations they access. At present, Polly focuses on changes of the execution order. Classical loop transformations such as interchange, tiling, fusion and fission but also advanced transformations [6] change the execution order. Further support is planned for optimizations that change data accesses [16, 19].

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 obvious, since only the schedule defines the execution times. However, some previous approaches [11] had difficulties to express certain transformations, e.g., tiling or index set splitting, without changes to the domain. Such difficulties do not arise in Polly, since the integer maps used to define the schedules are expressive enough to describe these transformations.

The schedules can be changed in two ways: either they can be replaced by schedules that are recalculated from scratch or they can be modified by a set of transformations. In Polly, a transformation is an integer map that maps from the

original 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 to the range of the first. To illustrate how transformations are performed in Polly, we present a loop blocking transformation for the following SCoP:

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

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

Loop blocking is a combination of the transformations $\mathcal{T}_{StripMine_{Outer}}$, $\mathcal{T}_{StripMine_{Inner}}$ and $\mathcal{T}_{Interchange}$.

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

Applying code generation for the statement with its domain $\mathcal{D}_{Stmt}$ together with the new schedule $\mathcal{S}'_{Stmt}$ yields the following code with blocked loops:

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

5.3. *External Optimizers—JSCoP*

Polly can export its internal polyhedral representation and reimport an optimized version of it. As a result, new optimizations can be tested without any knowledge of 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 is the scoplib format,[f] as presently used by Clan, Candl and PLuTo. The second is called *JSCoP* and is

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

specific to Polly. The main reason for defining our own exchange format is that scoplib is not expressive enough for Polly's internal representation: it does not support generic affine relations to describe the schedules or memory accesses that touch more than one element at a time. It also does not provide a convenient way to describe integer wrapping. 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 [8]. It contains a polyhedral description of a SCoP that follows the one used in scoplib.

We use the scoplib interface to connect PoCC as an external optimizer with Polly. This allows Polly to optimize the schedule of a SCoP automatically. The experiments in Section 7.2.2 and Section 7.2.3 show the sequential and parallel speedups gained.

6. From a Polyhedral Description to LLVM-IR

In Polly, transformations are applied to the polyhedral representation of a SCoP without changing the LLVM-IR of the program. Only after all transformations have been applied, LLVM-IR is regenerated and the actual program is updated. This section describes how we derive a generic AST from the polyhedral representation, the analysis we perform on the AST and how we use it to generate optimized LLVM-IR code. We discuss the generation of sequential, OpenMP and SIMD parallel code.

6.1. *Generating a Generic AST*

The first step from the polyhedral representation of a SCoP back to an imperative program is the construction of a generic, compiler-independent AST. The program described by this AST enumerates all statement instances in the order defined by the schedules. Bastoul [3] developed with CLooG a code generator that generates such an AST efficiently. CLooG uses an enhanced version of the Quilleré algorithm [21]. In Polly, we offload the construction of the generic AST entirely to CLooG.

6.2. *Analysis of the Generic AST*

Some analyses are better performed on the generic AST. The reason is that the polyhedral representation defines only the execution order of the statement instances, but not the exact control flow structures to execute them. Different ASTs may be generated for a single SCoP depending on the optimization goals chosen. Listing 3 shows two ASTs. The first has minimal code size: it does not contain duplicated statements. The second has minimal branching: all conditions are removed from the loop bodies. Still, both are generated from the following SCoP:

$$\mathcal{D}_{Stmt_1} = \{S1[i] : 0 \leq i \leq N\} \qquad \mathcal{D}_{Stmt_2} = \{S2[i] : 0 \leq i \leq 10\}$$
$$\mathcal{S}_{Stmt_1} = \{S1[i] \rightarrow [i, 0]\} \qquad \mathcal{S}_{Stmt_2} = \{S2[i] \rightarrow [i, 1]\}$$
$$\mathcal{A}_{Stmt_1} = \{S1[i] \rightarrow A[i]\} \qquad \mathcal{A}_{Stmt_2} = \{S2[i] \rightarrow B[0]\}$$

```
// Loop L1
for (i = 0; i <= N; i++) {
  A[i] = i;     // S1
  if (i <= 10)
    B[0] += i;  // S2
}
```

```
// Loop L2.1
for (i = 0; i <= min(10, N); i++) {
  A[i] = i;    // S1
  B[0] += i;   // S2
}

// Loop L2.2
for (i = 11; i <= N; i++)
  A[i] = i;    // S1
```

Listing 3: Two different ASTs generated from the same SCoP (left: minimal size, right: minimal control)

Analyses that derive information about the generated code, but that do not analyse the generated code itself, may be imprecise. Decisions taken during code generation are not available to such analyses. As a result, properties must be proven for the iteration space of the entire SCoP rather than just for the subspaces enumerated by individual loops. For example, loop $L_{2.2}$ in Listing 3 can be executed in parallel, but an analysis that does not take code generation into account cannot establish this.

To perform a precise analysis, we extract the subset of the schedule space that is enumerated by each loop. For example, for the loops in Listing 3, the following information is extracted:

$$\begin{aligned}\mathcal{E}_{L_1} &= \{[i,t] : 0 \leq i \leq N \wedge 0 < t \leq 1\}\\ \mathcal{E}_{L_{2.1}} &= \{[i,t] : 0 \leq i \wedge i \leq N \wedge i \leq 10 \wedge 0 < t \leq 1\}\\ \mathcal{E}_{L_{2.2}} &= \{[i,t] : 11 \leq i \leq N \wedge t = 0\}\end{aligned}$$

Even though this information is obtained after code generation, reparsing of the generated code is not necessary. Instead, the polyhedral information is exported directly from the code generator. Projects such as PoCC perform similar analyses, but they reanalyze the generated code. In contrast, Polly stays entirely within the polyhedral model.

6.2.1. *Detection of parallel loops*

Polly detects loops that can be executed in parallel as OpenMP or SIMD code. As explained in the previous section, this analysis must be performed after code generation.

Listing 4 shows code in which the instances of two statements are enumerated either in a single or in two separate loop nests. Which code is generated depends on the schedule of the statements. In the case of a single loop nest, no loop can be executed in parallel, since the i- and j-loops carry dependences of *S1* and the k-loop carries dependences of *S2*. However, in the case of two separate loop nests, each loop nest contains loops that do not carry any dependence. In the first loop

nest, these are the i- and j-loops and, in the second, the k-loop. All loops that do not carry any dependence can be executed in parallel.

To decide whether a certain loop, e.g., the innermost loop in the second loop nest of the non-fused code, is parallel, we check whether it carries any dependences. In our example, the dependences are $\mathcal{D} = \{S1[i,j,k] \to S1[i,j,k+1];\ S2[i,j,k] \to S2[i,j+1,k]; S2[i,-1+N,k] \to S2[1+i,0,k]\}$. $\mathcal{D}$ contains relations between statement instances. We translate them to the scheduling space by applying to their ranges and domains the statement schedules contained in $\mathcal{S} = \{S_1[i,j,k] \to [0,i,j,k]; S2[i,j,k] \to [1,i,j,k]\}$. The resulting dependences are $\mathcal{D}_S = \mathcal{S} \circ \mathcal{D} \circ \mathcal{S}^{-1} = \{[0,i,j,k] \to [0,i,j,k+1];\ [1,i,j,k] \to [1,i,j+1,k]; [1,i,-1+N,k] \to [1,1+i,0,k]\}$. $\mathcal{D}_S$ 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 $\mathcal{E}_{L_2} = \{[1,i,j,k] : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K\}$ such that the remaining dependences are $\mathcal{D}_{L_2} = \{[1,i,j,k] \to [1,i,j+1,k]; [1,i,-1+N,k] \to [1,1+i,0,k]\}$. If we now calculate the dependence distances with $deltas(\mathcal{D}_{L_2}) = \{[0,0,1,0]; [0,1,1-N,0]\}$, we can see that the second and the third dimension carry dependences while the others do not. Since the very first dimension does not depend on the induction variables, only the innermost dimension and, consequently, the innermost loop is parallel. Had we checked the innermost dimension before code generation, it would have carried a dependence and, consequently, parallel execution would have been invalid.

```
// fused → no parallelism
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++) {
S1:   C[i][j] += k;
S2:   X[k] += k;
    }
```

```
// non-fused → parallelism
for (i = 0; i < M; i++)     // parallel
  for (j = 0; j < N; j++)   // parallel
    for (k = 0; k < K; k++)
S1:   C[i][j] += k;

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

Listing 4: Different loop structures can hide or expose parallelism

6.2.2. *Memory access stride*

Especially for SIMD code generation, it is important to understand the pattern in which memory is accessed during the execution of a loop. Let us analyze one of these patterns: the stride, the distance between memory accesses in two subsequently executed statement instances. The analysis we describe is fully polyhedral and every loop in the generated AST is analyzed individually.

To calculate the stride of a specific memory access X, the following information is needed: the access relation $\mathcal{A}_X$ of access X, the schedule $\mathcal{S}$ of the statement

containing the memory access and the subset $\mathcal{E}$ of the schedule space enumerated by the loop we analyze. We illustrate this with the following example:

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

$$\begin{aligned}
\mathcal{A}_A &= \{S[i,j] \to A[j]\} \\
\mathcal{A}_B &= \{S[i,j] \to B[i]\} \\
\mathcal{A}_C &= \{S[i,j] \to C[2j]\} \\
\mathcal{S} &= \{S[i,j] \to [i,j]\} \\
\mathcal{E} &= \{[s_0, s_1] : 0 \leq s_0 \leq N \wedge 0 \leq s_1 \leq N\}
\end{aligned}$$

First, we create a map *NEXT* that maps from one iteration of the loop we consider to the iteration that follows immediately. This relation is computed from $\mathcal{S}$ and $\mathcal{E}$ by taking the lexicographic minimum of all the iterations that follow a given iteration. To find the distance between the array elements accessed in two neighboring iterations, we carry this relation over to the original domain $NEXT_{Dom} = \mathcal{S}^{-1} \circ NEXT \circ \mathcal{S}$ (so it relates iterations of the original loops before scheduling) and, finally, to the array indices $NEXT_X = \mathcal{A}_X \circ NEXT_{Dom} \circ \mathcal{A}_X^{-1}$. By calculating the difference between the array accesses in relation, we find the stride of the array access w.r.t. the loop considered: $Stride_X = deltas(Next_X)$.

In the example, set $\mathcal{E}$ contains both loops, i.e., the analysis is for the inner loop.

$$\begin{aligned}
NEXT &= \{[i,j] \to [i,j+1]\} \\
NEXT_{Dom} &= \{S[i,j] \to S[i,j+1] : 0 \leq i \leq N \wedge 0 \leq j < N\} \\
NEXT_A &= \{A[j] \to A[j+1] : 0 \leq j < N\} \\
NEXT_B &= \{B[i] \to B[i] : 0 \leq i \leq N\} \\
NEXT_C &= \{C[2j] \to C[2j+2] : 0 \leq 2j \leq N-2\}
\end{aligned}$$

Hence, the access distances (w.r.t. the j-loop) are 1, 0 and 2, respectively for the three accesses.

6.3. *Generation of LLVM-IR Code*

Polly regenerates LLVM-IR using the generic AST as a description of the new program structure. By default, Polly generates sequential code but, if requested, it generates OpenMP and SIMD code to take advantage of parallelism.

6.3.1. *Sequential code generation*

To generate sequential code, Polly replaces abstract FOR loops with sequential LLVM-IR loops and abstract IF statements with LLVM-IR conditional branches. The newly generated loops introduce new induction variables. Code for abstract

statements is generated by copying the corresponding basic block. Since we make certain that all basic blocks are independent (Section 4.4.2), generating code for them is simple. For each block, we generate code that calculates the values of the old induction variables as a function of new induction variables and parameters. Then, all instructions are copied from the old block and their operands are updated. Operands pointing to an old induction variable are switched to their recalculated value and operands pointing to a value calculated in the same basic block are switched to the result of the copied instruction. The remaining operands refer to parameters and are not changed. Branch instructions and ϕ-nodes are not copied. Branch instructions are replaced by newly generated control flow. The only ϕ-nodes that can exist in an independent block are loop induction variables. Since these are explicitly code generated, the original ϕ-nodes can be ignored.

After code generation is finished, there is no need for independent blocks in the CFG. Consequently, we can reintroduce scalar dependences to eliminate the redundant scalar computations. We do this by running the scalar optimizations available within LLVM. Section 7.2 shows that this removes the previously introduced overhead reliably.

6.3.2. *OpenMP code generation*

Polly can take advantage of thread-level parallelism by generating code for the GNU OpenMP run-time system. If requested, it transforms every parallel loop (see Section 6.2.1) that is not surrounded by another parallel loop into an OpenMP parallel loop. The generated code is run-time configurable, such that the user can define the scheduling policy and the number of execution threads by setting special environment variables. Aloor [1] describes in detail how Polly generates OpenMP code.

6.3.3. *Vector code generation*

Polly generates vector code for trivially vectorizable loops. A loop is trivially vectorizable if it is parallel (Section 6.2.1), has a constant, small, non-parametric number of loop iterations and does not contain conditional control flow or any further loops. Listing 5 shows (a) a loop, (b) a version of it that exposes a trivially vectorizable loop, and (c) the vectorized loop.

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

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

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

Listing 5: Three steps to vectorize a loop

The vector code generation of Polly is on purpose limited to a very restricted set of loops in order to decouple the actual vector code generation from the transformations that enable it. Preparing optimizations are required to expose trivially vectorizable loops, which the Polly code generation detects and transforms to platform-independent vector operations. Then, the LLVM back ends translate the platform-independent vector instructions to efficient platform-specific operations.

When generating vector code for a loop body, Polly creates, for each original instruction, a set of scalar instructions, one for each loop iteration. Vector code is introduced as soon as a load instruction is reached. We generate scalar loads for the different instances of the load instruction and, in addition, instructions that combine these scalars to a vector. Any later instruction in the original loop that uses the loaded scalar, or a value derived from it, is translated to a vector instruction, such that all instances of the original instruction are executed in a single vector operation. Starting from a load, all calculations depending on the loaded value are performed 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.

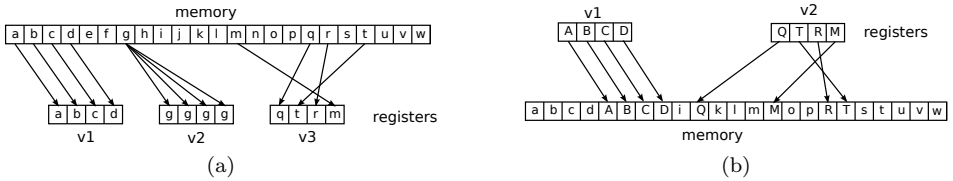


Fig. 4. (a) Three kinds of vector loads: (v1) stride-one load, (v2) stride-zero load, (v3) complex; (b) Two kinds of vector stores: (v1) stride-one store and (v2) complex store

Even though we use vector instructions for the actual calculations, the memory access itself is still performed via scalar operations. To optimize this, we calculate the strides of the memory accesses (Section 6.2.2). There are three types of loads: stride-one loads, stride-zero loads and complex loads. There are two types of stores: stride-one stores and complex stores; see Figure 4. Polly optimizes all except the complex accesses. It implements a stride-zero load by loading the scalar value once and copying it to the individual vector elements. For stride-one vector accesses, it uses whole vector load and store operations. Further optimizations are possible starting from negative stride-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.

There are further requirements for high-performance vector code. To enable the LLVM back ends to issue optimal vector instructions, the alignment of load and store instructions must be specified. LLVM can often calculate this information during its standard optimization passes. In case it cannot, we plan to derive this information with Polly.

7. Experiments

7.1. *Guided Vectorization*

Dense matrix multiplication (gemm) is an important computation. Optimizing it well is crucial. We have optimized a challenging variant of gemm and found notable improvements over current compilers. The first kernel in Listing 6 shows the 32x32 gemm kernel we optimized. In this kernel, memory accesses have unit stride in all loops. If the kernel is vectorized along one loop, only one loop has unit stride; the other two loops have non-unit strides. Clever use of unrolling, loop interchange and strip mining reduces the cost of the non-unit stride accesses such that vectorization is beneficial.

```
// 1. Original kernel
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];
```

```
// 2. Kernel prepared for vectorization
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];
```

```
// 3. Vectorized kernel
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 6: Three steps to vectorize a challenging gemm kernel

Figure 5 shows the run times we measured. The baseline is LLVM 2.8 with all optimizations enabled. Both LLVM and GCC 4.4.5 do not change the loop structure and create no SIMD instructions. Differences exist in the scalar optimizations. Here, LLVM is more effective such that its code runs faster. ICC 11.1 performs loop transformations and introduces SIMD instructions. Its code is almost twice as fast as the one generated with LLVM. Yet, it still requires a large number of scalar loads, which suggests that further optimization is possible.

Then, Polly was used to generate optimized vector code. We tried Polly: Only LLVM -O3 first, which runs Polly but does not apply any transformations. The unchanged run time shows, that the indirection via Polly does not add any overhead. Polly: +Strip mining changes the loop structure to improve data locality and to expose a trivially vectorizable loop. Stock [23] obtained the improved structure via iterative compilation. We derived a polyhedral schedule manually from the improved structure and imported the schedule into Polly. The second kernel in Listing 6 shows the new loop structure. It contains an innermost, trivially vectorizable jj-loop. In addition, the i-loop was moved to a deeper loop level. This simple loop structure

change increases the performance by 19%.

Polly: += Vectorization replaces the previously created trivially vectorizable loop with SIMD instructions. Polly recognizes that the innermost loop is parallel and performs the necessary transformations automatically. Following Section 6.3.3, Polly generates full vector loads for the access to C, a stride-zero load for the access to A, and scalar loads for the elements loaded from B. At this point, the Polly-optimized code is twice as fast as the LLVM base line and the performance of ICC is reached.

The load from B in the innermost loop is still inefficient as it requires four scalar loads to initialize the vector. However, due to our previous changes, the loads from B are invariant in the innermost loop. Since Polly can establish that the i-loop is executed at least once, the loads of B can be hoisted out. Polly: += Hoisting shows that this transformation triples the performance. Now, Polly outperforms ICC easily.

Finally, it is possible to trade code size for performance. By increasing the unrolling limits, the inner two loops can be fully unrolled. This leads to further increases in performance. The overall speedup we achieve is 8x compared to plain LLVM and 4x over ICC.

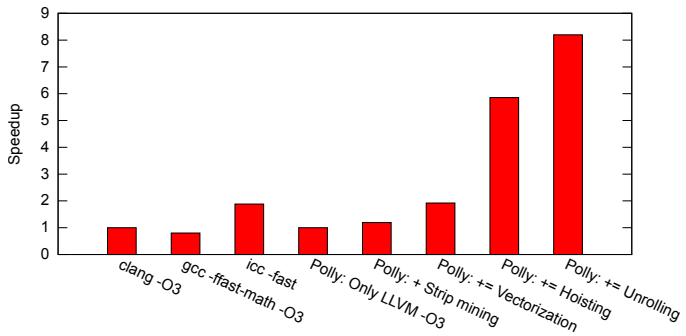


Fig. 5. 32x32 single precision gemm on Intel® Core™ i5

We have shown that Polly can improve performance significantly in comparison with ICC and Clang/LLVM. All transformations were performed automatically within Polly. The only manual component here is the externally provided schedule. We conclude that, with Polly, data access optimizations can be applied effectively to a low-level program by simply providing a polyhedral schedule. It is no longer necessary to understand low-level internals, but one is free to focus on the high-level optimization problem.

7.2. *Automatic Optimization*

To understand the impact of Polly, we analyzed the run time of the PolyBench 2.0 benchmark suite,[g] which contains computation kernels used in linear algebra, data mining, stencil and image processing. We conducted all experiments on the unmodified[h] C source code, where Polly detects the SCoPs automatically without using any source code annotations. All tests have been run on a 2-socket, 12-core, 24-threads Intel® Xeon® X5670 system with double-precision floating-point arithmetic.

7.2.1. *The identity transformation*

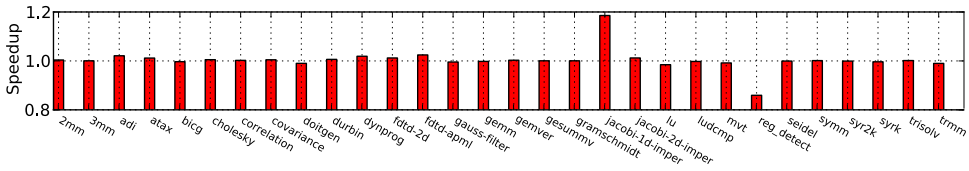


Fig. 6. Run time of Polly-optimized code

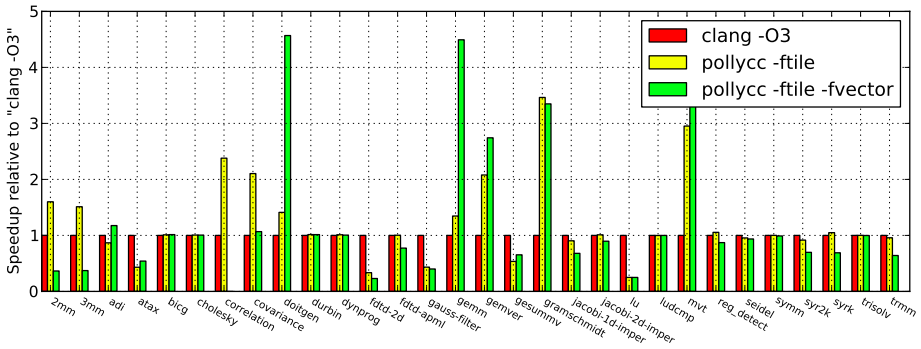
To estimate the overhead Polly introduces, we analyze the identity transformation. The identity transformation translates from LLVM-IR to the polyhedral representation and back to LLVM-IR, but does not apply any polyhedral transformations. The results in Figure 6 show only for two benchmarks large run time changes. The remaining 28 benchmarks show less than 2% difference. The two large changes are a 25% slowdown on reg_detect and an 18% speedup on jacobi-1d-imper. Both are due to optimizations in LLVM, which trigger due to different code structure only on the new or the old code, but not on both. The only two performance regressions are caused by missed optimizations within LLVM. The regeneration of LLVM-IR by Polly does not introduce any significant overhead.

7.2.2. *Sequential code*

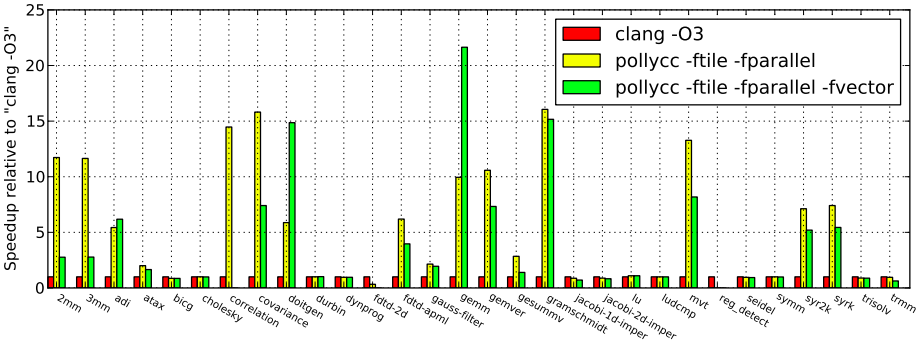
Let us investigate the effects of automatic optimizations on sequential performance using PoCC as external optimizer. The PoCC optimizer uses the PLuTo algorithm to maximize data locality and tileability. All benchmarks were run for both small and large data sizes.

The results for the small data size are shown in Figure 7a. Seven benchmarks exhibit a decrease in performance. Here, either tiling is not beneficial because of limited data reuse or the problem size is too small to benefit from tiling. The decreased performance is expected, since Polly does not yet include a profitability heuristic

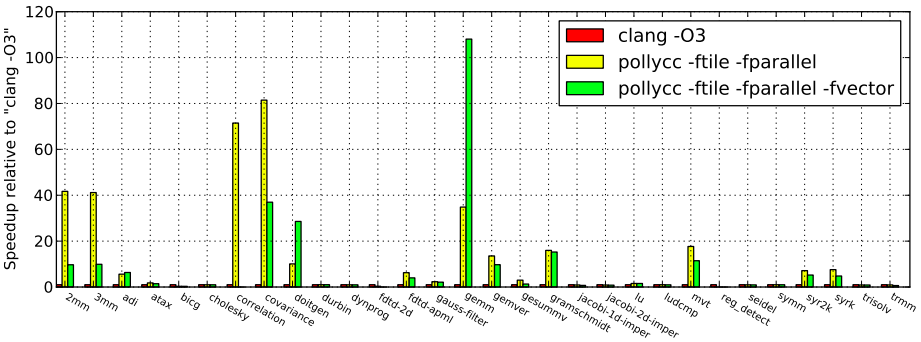
[g] `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`
[h] The type of some induction variables is changed to remove currently unsupported implicit casts.



(a) Sequential, small data size



(b) Parallel with 24 threads, small data size



(c) Parallel with 24 threads, large data size

Fig. 7. Speedup of pollycc compared to clang (both current from 25/03/2011) on Intel® Xeon® X5670

for tiling. For nine of thirty benchmarks, we see large performance improvements even on small data sizes. For larger data sizes (not shown), we obtain on average a

2x speedup with four benchmarks showing more than a 4x speedup and two even reaching an 8x speedup. This is promising, since we have not yet analyzed benchmarks individually. SIMD code generation improves the run time in two cases even further, with gemm reaching an overall speedup of 14x. Since Polly and PLuTo do not yet specifically optimize for SIMDization, SIMDization also causes reduced performance in several cases. This can be improved by including SIMDization in polyhedral scheduling [26].

7.2.3. *Parallel code*

Let us also analyze the speedups we obtain with OpenMP parallel code. We used PLuTo to expose parallelism and Polly to create OpenMP code that takes advantage of the exposed parallelism.

Figure 7b shows that, for the small data size, sixteen of thirty benchmarks benefit from parallel execution. Eight benchmarks reach more than a 10x speedup. For the larger data sizes, the speedups are shown in Figure 7c. We obtain on average a 12x speedup and there is even a case with a more than 80x speedup. With additional SIMD code generation, the gemm kernel is again the top performer with an overall speedup of more than 100x. It can be observed that parallelism combined with tiling and vectorization can show significant performance improvements for a large number of benchmarks. We expect further improvements by analyzing the benchmarks individually. Also, additional benefits through the implementation of vectorization heuristics are expected.

8. Future work

Even though Polly implements the polyhedron model on LLVM-IR, there are several areas where it can be enhanced. To preserve the program semantics when arithmetic overflows in the affine expressions (before or after transformation and code generation) occur during execution, work is necessary in both Polly and LLVM itself to improve the handling of possible integer overflows.

Several improvements can be made to increase the number of codes that can be handled by Polly. Modeling the memory behaviour of certain function calls and intrinsics, especially calls to memcpy, memove and memset, seems worthwhile. Adding support for cast and modulo operations in the affine expressions is possible since isl supports modulo arithmetic. Dynamically allocated multi-dimensional arrays are represented as one-dimensional arrays with non-affine subscripts (e.g., $A[n \cdot i + j]$ instead of $A[i][j]$ with n being the size of the inner dimension). To make these codes amenable to polyhedral optimizations in Polly, a recovery of dynamic multi-dimensional arrays will be needed. Beyond these static techniques, non-static techniques [4] may be used to increase the amount of code further that Polly can optimize.

At present, Polly cannot modify the structure of the basic blocks (which it treats as the statements in the polyhedral representation). To enable optimizations

at a finer granularity, we are working on splitting basic blocks into several statements (when possible) to expose more potential for parallelism. In addition, we have started work on allowing modifications of the data layout (i.e., the access functions).

Finally, we plan to integrate a polyhedral optimizer directly into Polly. Due to the promising results with PLuTo, an enhanced version of the PLuTo algorithm was added to isl. Preliminary support from within Polly exists, but testing for robustness, stability and scalability is still required.

9. Conclusion

With Polly, we offer a polyhedral optimizer for a low-level intermediate representation. We have shown how to extract relevant program parts, how to translate them into a polyhedral representation, how to apply optimizations and, finally, how to generate optimized program code. The process described is not bound to a specific high-level programming language and does not require the input code to exhibit any syntactic format. As a result, constructs such as GOTO loops or pointer arithmetic can be optimized.

We use a Z-polyhedral representation (based on isl integer sets) that can either be analyzed and optimized in Polly or exported to external optimizers. With PoCC, we have integrated an external optimizer which provides the PLuTo optimization of data locality while, at the same time, maximizing the exposed parallelism. After the code has been optimized, Polly detects available parallelism automatically and generates optimized OpenMP or SIMD code. This allows optimizers to focus on the parallelisation problem itself and to offload low-level details to Polly.

To demonstrate the effectiveness of Polly, we optimized the PolyBench 2.0 test suite and analyzed the results. The detour via the polyhedral representation does not introduce any overhead in the generated code. In the case that PLuTo optimizations are used, we have observed significant speedups that reached 14x without thread-level parallelism and more than 100x with thread-level parallelism. For larger data sizes, we reached on average a 2x speedup without and a 12x speedup with thread-level parallelism. The speedups show clearly that Polly can effectively apply generic, high-level transformations directly to low-level code. This increases the abstraction level, at which compiler optimizations can be developed, notably.

We hope that Polly will be an interesting tool for the development of new optimizations and for their evaluation on real applications. In addition to targeting multicore systems, we see Polly's potential also as an integrated optimizer that targets heterogeneous platforms consisting of tightly integrated CPUs and accelerators. For example, due to the high number of OpenCL implementations based on LLVM (AMD, NVIDIA, Intel, Apple, ...), Polly and LLVM may be a convenient platform for research on polyhedral optimization in this area.

Acknowledgements

Thanks go to Hongbin Zheng, Raghesh Aloor and Andreas Simbürger for their work on Polly. We are grateful for the advice and academic work of Albert Cohen, Martin Griebl, Sebastian Pop, Louis-Noël Pouchet, and Sven Verdoolaege, who influenced Polly directly and indirectly. Special thanks go to Dirk Beyer who supported the development of the RegionInfo analysis with several university projects and, in particular, to P. Sadayappan, who supported part of this work generously via a research scholarship at Ohio State (NSF 0811781 and 0926688). This work was also sponsored in part by a Google Europe Doctoral Fellowship in Efficient Computing.

References

[1] R. Aloor. A framework for automatic OpenMP code generation. Master's thesis, IIT Madras, 2011.

[2] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In R. Gupta, editor, *Compiler Construction (CC 2010)*, LNCS 6011, pages 244–263. Springer-Verlag, 2010.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004)*, pages 7–16. IEEE Computer Society, 2004.

[4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In R. Gupta, editor, *Compiler Construction (CC 2010)*, LNCS 6011, pages 283–303. Springer-Verlag, 2010.

[5] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2010)*, pages 343–352. ACM, 2010.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2008)*, pages 101–113. ACM, 2008.

[7] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[8] D. Crockford. The application/json media type for JavaScript object notation (JSON). RFC 4627 (Informational), July 2006.

[9] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.

[10] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[11] 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. *Int. J. Parallel Programming*, 34(3):261–317, June 2006.

[12] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 603–604. Springer-Verlag, 1997.

[13] T. Grosser. Enabling polyhedral optimizations in LLVM. Master's thesis, University of Passau, 2011.

[14] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly—Polyhedral optimization in LLVM. In *Proc. First Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2011)*, Apr. 2011.

[15] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proc. 5th Int. Conf. on Supercomputing (ICS'91)*, pages 244–251. ACM, 1991.

[16] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A hyperplane-based approach for optimizing spatial locality in loop nests. In *Proc. 12th Int. Conf. on Supercomputing (ICS'98)*, pages 69–76. ACM, 1998.

[17] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, Aug. 2003.

[18] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *Proc. IEEE First Int. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP'95)*, pages 153–162. IEEE, 1995.

[19] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. Supercomputing*, 21(1):37–76, Jan. 2002.

[20] 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. A. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'93)*, LNCS 768. Springer-Verlag, 1993.

[21] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Programming*, 28(5):469–498, Oct. 2000.

[22] T. Risset, S. Derrien, P. Quinton, and S. Rajopadhye. High-level synthesis of loops using the polyhedral model. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis : From Algorithm to Digital Circuit*, chapter 12. Springer-Verlag, 2008.

[23] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison. Model-driven SIMD code generation for a multi-resolution tensor kernel. In *Proc. IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2011)*, pages 1058–1067. IEEE Computer Society, 2011.

[24] 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 *Proc. 2nd Int. Workshop on GCC Research Opportunities (GROW 2010)*, pages 1–13, Jan. 2010.

[25] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In J. Knoop, editor, *Compiler Construction (CC 2001)*, LNCS 6601, pages 118–132. Springer-Verlag, 2001.

[26] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2012)*, Jan. 2012.

[27] S. Verdoolaege. isl: An integer set library for the polyhedral model. In A. Iglesias and N. Takayama, editors, *Mathematical Software (ICMS 2010)*, LNCS 4151, pages 299–302. Springer-Verlag, 2010.

[28] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.

[29] B. Yee, D. Sehr, G. Dardyk, et al. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. 30th IEEE Symposium on Security and Privacy (S&P 2009)*, pages 79–93. IEEE Computer Society, 2009.