The Potential of Polyhedral Optimization: An Empirical Study

Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer
University of Passau, Germany

***Abstract*—Present-day automatic optimization relies on powerful static (i.e., compile-time) analysis and transformation methods. One popular platform for automatic optimization is the polyhedron model. Yet, after several decades of development, there remains a lack of empirical evidence of the model's benefits for real-world software systems. We report on an empirical study in which we analyzed a set of popular software systems, distributed across various application domains. We found that polyhedral analysis at compile time often lacks the information necessary to exploit the potential for optimization of a program's execution. However, when conducted also at run time, polyhedral analysis shows greater relevance for real-world applications. On average, the share of the execution time amenable to polyhedral optimization is increased by a factor of nearly 3. Based on our experimental results, we discuss the merits and potential of polyhedral optimization at compile time and run time.**

I. Introduction

Automatic code optimization is becoming an increasingly challenging task. The variety and complexity of optimization targets have increased recently, with the introduction of hyperthreading, SIMD extensions, multicore processors, general-purpose computing on graphics hardware (GPGPU computing), low-power computing, etc. Programmers and compiler implementers are being confronted with the architectural differences and, consequently, the non-portability of performance between different platforms. Just setting a few compiler optimization flags is no longer sufficient.

To tackle this problem, a wide variety of approaches to the *automatic* optimization of program code has been proposed [1], [2], [3], [4]. One popular approach is *polyhedral optimization*. It is based on the *polyhedron model* [1], which represents iterative executions as polyhedra. The model serves to optimize automatically programs containing loops (e.g., via parallelization or data localization) by applying algebraic transformations. Its main benefit is that all loop transformations can be discovered via linear programming and at a cost that is independent of the problem size.

Full automation comes at the price of limitations on the structure of the programs that can be optimized. In particular, loop bounds and memory-access functions are limited to affine linear expressions, in order to be analyzable at compile time (e.g., array access functions like `A[3*i+1]` are allowed, whereas access functions like `A[i*i]` are disallowed). In the past, a number of extensions of the polyhedron model have been proposed to overcome affine linearity in certain cases [5]. Another promising approach is to apply polyhedral optimization not only at compile time but also at run time. The idea is that, at run time, more is known about the actual structure of the loops, thus enabling more loops to be optimized. Among the many implementations of polyhedral optimizers, there are two major projects that target main-stream compilers: Polly [6], [7] and Graphite [8]. Both share the approach of an automatic and language-agnostic detection of compatible loops at compile time, called *static control parts* (SCoPs).

Despite the rich work on extending the polyhedron model and polyhedral optimization, there is a lack of empirical evidence of the relevance of the model and its variants in practice across different domains. Therefore, for the first time, we conducted a comprehensive empirical study on the practicality of polyhedral optimization for automatic loop parallelization. In particular, we are interested in (1) whether state-of-the-art polyhedral methods are applicable and beneficial in practice, and (2) how far this potential can be enhanced by an application also at run time.

In an empirical study, we analyzed a corpus of 51 real-world C/C++ programs, using (an extension of) the plugin Polly for the LLVM compiler infrastructure [9]. These subject programs are of different sizes (500–600 000 lines of code) and different domains (e.g., multimedia processing, compression, scientific computing, and compilers). In particular, we measured the fraction of the program code that can be expressed in the polyhedron model and the time spent inside static control parts (SCoPs) in relation to the total run time. This approach is more general than quantifying the effects of a certain specific instance of polyhedral optimization, because it provides a perspective on the applicability of an entire collection of optimizations in the polyhedron model.

We found that the benefit gained by state-of-the-art compile-time polyhedral optimization is rather marginal (10 % on average). This is mainly due to a lack of knowledge of parameter values at compile time, which may lead to non-linearity in the memory access functions of a loop program. The potential benefit of run-time optimizations is higher. Just by supplying the missing information via run-time program analysis, the SCoP coverage could be raised from 4 % to 41 % (29 % on average).

Based on our findings, we discuss, for the first time, the merits of polyhedral optimization in practical programming. We conclude that it is challenging to apply polyhedral optimization to the most critical regions of a program without additional knowledge about the program's execution. However, by exploiting information available *just in time*, the polyhedron model is capable of covering most of the critical program regions and providing program transformations to exploit the potential for parallelism and other optimizations.

978-1-4799-0214-9/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA
Experience Track

Our findings suggest that future research should focus on acquiring more information about the critical program regions at run time. This would increase the impact of polyhedral optimization in real-world applications.

In summary, we make the following contributions:

- the first substantial empirical study of the impact of polyhedral optimization across a broad spectrum of applications,
- an open-source polyhedral analysis engine that is independent of the syntax of the source language and that incorporates extensions of the polyhedron model for inclusion of run-time information in the optimization,
- a novel instrumentation-based measurement method that quantifies the dynamic coverage that polyhedral optimizations can attain,
- a discussion of the practical potential and perspectives of compile-time and run-time polyhedral optimization based on the empirical results and a statistical analysis.

The analysis engine, the sample programs, and all results of our empirical study are available at the project's Web site:
http://www.infosun.fim.uni-passau.de/cl/staff/simbuerger/pprof/

II. The Polyhedron Model

In this section, we introduce the necessary background of the polyhedron model. The polyhedron model represents programs –in particular, loops– in an algebraic form as polyhedra, to make them amenable to algebraic transformations. A major use case of the model and the corresponding transformations is to parallelize loops; others are cache-locality optimization, and memory-usage optimization (see Sec. VI). The main benefit of the polyhedron model is that loop transformations can be derived fully automatically and independently of the problem size. The price paid to attain full automation and feasible complexity is that not all kinds of programs can be processed.

The polyhedral optimization of a program consists of two steps: (1) detecting the loops of a program that can be represented in the model, called *static control part*s (SCoPs) [10], and (2) applying the actual transformations to optimize the program (loop parallelization, etc.).

In the remainder of the section, we introduce the basic model and classify SCoPs as compile-time (*Static*) or run-time (*Dynamic*). A comprehensive survey of the polyhedron model can be found elsewhere [1].

A. Basic Model

In the polyhedron model, a loop program consists of a number of statements. Each statement has an associated iteration domain (which is defined by the loops surrounding the statement) and a schedule (which determines the execution order of the statement instances). In the following, we will use Figure 1, which gives an example of a SCoP, to introduce briefly the key concepts of the polyhedron model.

Each program statement S comes with its own iteration domain D_S, which is a subset of $\mathbb{Z}^n$. Each point $\vec{i} \in D_S$ in this domain represents a statement instance $(S;\vec{i})$, i.e., statement S is executed once for every such $\vec{i}$. Our example has two statements: S and T (see Lines 4 and 5). They are

```
1    for (int i=0; i<=n; ++i)
2      for (int j=i; j<=n; ++j)
3        if (i >= n-j) {
4  S:     A[i+n][j+i] = B[n+2*i-1][j];
5  T:     B[i+n][j-i] = A[n-2*i+1][j];
6        }
```

Fig. 1: A static control part (SCoP)

surrounded by two loops and an **if** statement and, therefore, share the iteration domain $D_S = D_T = \{[i,j] : 0 \leq i \leq n \ \wedge \ i \leq j \leq n \wedge n - j \leq i\}$, where n denotes a structure parameter that is constant during the execution of S and T. Note that the control flow is known at compile time because the predicate and the loop bounds are affine expressions.

Each statement can contain memory *accesses* to arrays (the aggregate data structure the model focusses on); these are represented by relations between the domain of the statement and the indices of the array cells accessed. In our example, both statements perform a read access and subsequently a write access. The read and write accesses are summarized in the relations R and W, respectively:

$$\begin{aligned} R = \{&S[i,j] \mapsto B[n+2*i-1,j]; \\ &T[i,j] \mapsto A[n-2*i+1,j]\} \\ W = \{&S[i,j] \mapsto A[i+n,j+i]; \\ &T[i,j] \mapsto B[i+n,j-i]\} \end{aligned}$$

R and W map the iteration (i,j) of S and T to the elements of A and B, respectively, which are accessed.

Transformations of the program must not violate the program's semantics by executing dependent statement instances in the wrong order or in parallel. Therefore, the most important computational task, when using the polyhedron model for program transformations, is to compute the dependences between statement instances. A statement instance $(T;\vec{j})$ depends on an instance $(S;\vec{i})$ iff there are memory-accesses to the same memory cell in S and T (for the given values of $\vec{i}$ and $\vec{j}$).

Determining the dependences is undecidable in the general case, because this would require to solve arbitrary systems of equations derived from the accesses (cf. the unsolvability of Hilbert's 10^{th} problem [11]). However, dependences can be computed when iteration domains and accesses are defined by affine expressions (i.e., all constraints can be written in the form $M\vec{i} \geq \vec{b}$ for $M \in \mathbb{Z}^{k\times n}$, $\vec{b} \in \mathbb{Z}^k$).[1] Note that some dimensions of $\vec{i}$ can be structure parameters, which allows parameters to occur additively (*weak* parametrization; Figure 2a) in all constraints, schedules and memory accesses, but not multiplicatively (*strong* parametrization; Figure 2b).

The restriction to affine expressions implies that non-affine conditions and recursive control flow cannot benefit from polyhedral optimization. Other consequences of the restriction are that the only aggregate data structure allowed is the array (scalars can be represented as zero-dimensional arrays), and the

[1] Geometrically, these objects are ($\mathbb{Z}$-)polyhedra.

only statement type allowed in the loop body is the assignment. Calls of functions with side effects inside a loop body are not supported by the basic model, because the memory access behavior and the control flow are hidden inside the body of the function called.

Clearly, these strict requirements limit the number of programs that can be analyzed automatically. The key question of how many SCoPs are amenable to the polyhedral optimization of practical programs, and to which extent, is the driving motivation of our empirical study (see Sec. III).

B. Classification of SCoPs

The successful detection of SCoPs in given source code depends on two factors. First, is the detection performed at compile time or at run time? Second, which extensions are applied to the model to overcome certain restrictions, e.g., extensions to deal with multiplicative parameters via quantifier elimination in the reals [5]?

Let us introduce two different classes of SCoPs –*Static* and *Dynamic*– which we address in our empirical study.

1) Class Static: This class covers all SCoPs that can be represented in the basic polyhedron model and to which all corresponding analysis and transformation steps can be applied at compile time.

2) Class Dynamic: Much like class *Static*, this class incorporates also only SCoPs that can be represented in the basic polyhedron model. However, the SCoP detection takes place while the program to be analyzed is being executed.

Class *Static* reflects the current state of the art of polyhedral optimization. At compile time, any possible violation of the model's restrictions in a given part of the code is prohibitive. But, the additional knowledge available at run time enables a more precise evaluation of the adherence to the restrictions in the given program run. Class *Dynamic* encompasses all code regions that are SCoPs when the values of parameters and the aliasing of pointers and arrays are known. In addition, control flow may be amenable to analysis at run time. Hence, *Dynamic* covers a larger set of SCoPs than *Static*.

Current implementations of the polyhedron model (e.g., POLLY or GRAPHITE) are capable of detecting SCoPs of class *Static* only. Thus, we had to prepare our experimental setup to be able to detect SCoPs of classes *Dynamic* (see Sec. III-H).

Next, we describe three SCoP patterns that fall into class *Dynamic*, but not into class *Static*. We use these patterns in our empirical study.

a) Known Parameters: The basic polyhedron model requires all loop bounds and memory accesses to be affine linear (Figure 2a, weak parametrization). Non-linearity introduced by parameters, as shown in Figure 2b (strong parametrization), cannot be handled in the basic model, i.e., is not in *Static*.

In Figure 2b, parameter `m`, although loop-invariant, is multiplied with the value of iteration variable `i`, forming a non-linear expression. However, the value of parameter `m` is known at run time. By substituting it for the parameter name, the loop nest complies with the polyhedron model.

```
1 int i;
2 for (i=0; i<=n; i++) {
3   A[i+n] = __;
4   __ = A[i-1+n];
5 }
```

(a) linear memory access (weak parametrization)

```
1 int i;
2 for (i=0; i<=n; i++) {
3   A[m*i+n] = __;
4   __ = A[m*(i-1)+n];
5 }
```

(b) non-linear memory access (strong parametrization)

Fig. 2: Linear vs. non-linear memory access. The expression $i-n+1$ can be handled in the basic polyhedron model. The expression $m*(i-1)+n$ cannot be handled in the basic polyhedron model, due to the multiplicative parameter m. The wildcard `__` can be replaced by any expression.

Run-time knowledge about parameters is not limited to constant parameter values. If parameter `m` adopts a limited number of values, one can provide a specialized loop code for each value. In the worst case, polyhedral analysis and optimization has to be performed every time the loop nest is reached by the control flow of the program.

b) Known Aliasing: As soon as we consider input languages that support pointers, we have to deal with the possibility of aliasing. Contemporary alias analyses can provide only a conservative approximation, leading to a so-called *may-alias*, i.e., an alias whose existence must be assumed but is uncertain. There are two alternative ways of dealing with a may-alias: (1) one postulates the corresponding dependence at compile time or (2) one tests for the alias at run time and respects its dependence conditionally.

In class *Dynamic*, we rely on the fact that the actual aliasing is revealed at run time and, therefore, we include SCoPs that give rise to unknown aliasing behavior at compile time.

c) Known Control Flow and Side Effects: Aside from the restrictions on loop bounds and arrays, the polyhedron model requires a well-formed control flow of the loop nest: any conditional in its body must also be of affine linear form $Ax \geq b$ (e.g., '**if** `(i >= n-j){ ... }`' is in affine linear form, while '**if** `(i >= random()){ ... }`' is not), and any side effects of function calls must be known (*must*-alias information –i.e., an alias whose (non-)existence is certain– of the callee must be available).

With run-time information, it becomes possible to establish the loop invariance or affine linearity of certain conditional predicates using the known parameter values. In the case of a function call with unknown side effects, there is no possibility of a reasonable polyhedral analysis at compile time: one would have to view the entire body of the respective function call as atomic with an arbitrary effect on the entire memory.

In class *Dynamic*, we permit calls of functions whose bodies form valid SCoPs at compile time, i.e., the function body must be in class *Static* and conditionals must become affine linear after known parameter values have been substituted.

In class *Dynamic*, we have to be careful which functions we allow to be called in SCoPs. Unfortunately, calling a function that itself forms a valid SCoP in class *Dynamic*, cannot be permitted in general. The function may use its

arguments (parameters) in products with iterators, and calling the function with an iterator as argument leads to non-linearities. However, we can permit functions that form valid SCoPs in class *Static*, since the function arguments can only occur as linear parameters in the contained SCoP. We restrict the arguments of the function call to affine linear expressions. This restriction is conservative, since the uses of a called function's parameters determine the compatibility with the model.

III. Experiment Setup and Execution

We conducted an empirical study to estimate the potential of polyhedral optimization of real-world programs.

A. Goals

We designed our study to answer the following questions. What is the potential of the basic polyhedron model at compile time? Can the applicability of polyhedral optimization be improved by using run-time information? How beneficial is the polyhedron model if run-time information is added?

B. Measurement Methodology

Empirical evaluations in the context of polyhedral optimization proceed usually by measuring the effect of a specific instance of polyhedral optimization on the run time of benchmarks. We have a more general view on this issue and do not limit our focus to a specific optimization. We are interested in the fraction of the run time that is, *in general*, in reach of polyhedral optimization. Therefore, instead of measuring the run time of a transformed program, we analyze the fraction of the sequential run time that we can reach with polyhedral transformation, the *execution SCoP coverage*. This requires a novel instrumentation approach (see Sec. III-H).

Definition (Execution SCoP coverage)**.** *Let $\mathbb{S}$ be the set of SCoPs of a program. Let $t : \mathbb{S} \to \mathbb{R}$ be a function that returns the accumulated run time of a SCoP in the run(s) of the program. Let T be the accumulated run time of the program for arbitrary sets of input values.* Execution SCoP coverage *(ExecCov) is then defined as:*

$$\textit{ExecCov} := \frac{1}{T} * \sum_{s \in \mathbb{S}} t(s)$$

Execution SCoP coverage allows us to estimate the potential benefit of an optimization by determining the fraction of the sequential program's run time that is spent inside SCoPs. Furthermore, execution SCoP coverage shows whether our transformations were able to hit the *hot spot(s)* of the program. The higher the SCoP coverage, the larger the impact of polyhedral optimization on a specific program run.

Our experiments measure the execution SCoP coverage of each program in the two classes *Static* ($\textit{ExecCov}_{\text{Stat}}$) and *Dynamic* ($\textit{ExecCov}_{\text{Dyn}}$). Note that, even though *ExecCov* can be ordered by $\textit{ExecCov}_{\text{Stat}} \leq \textit{ExecCov}_{\text{Dyn}}$, this does *not* imply that *Static* $\subseteq$ *Dynamic*.[2]

[2]Every SCoP detected is maximized in size, i.e., consecutive SCoPs are merged to one single SCoP. Therefore, a SCoP in *Static* does not necessarily correspond to the same SCoP in *Dynamic*. However, every statement that is part of a SCoP in *Static* is part of a SCoP in *Dynamic*.

C. Experiment Variables

The experimental variables of our empirical study are listed in Table I. We consider only one dependent variable: *ExecCov*. It varies depending on the two independent variables *Class* and *Dom*. Of course, we expect that the choice of using compile-time information only (*Static*) or run-time information additionally (*Dynamic*) during SCoP detection influences *ExecCov*.

Furthermore, we consider the domain to which a subject program belongs an independent variable (*Dom*). The reason is that we expect that some domains are more amenable to polyhedral optimization (e.g., scientific code) than others (e.g., database systems).

ExecCov is influenced not only by the SCoP class and domain but also by input data that we pass to the program being executed (i.e., the benchmark we use). These input data have an effect on the code paths chosen in the program run and, therefore, influence the fraction of SCoPs that are executed during a program run. We controlled this variable (*Input*) by making it constant per program. Nevertheless, we took care that the benchmarks we selected are realistic and cover a major fraction of the code paths in the program (see Sec. III-F). Further ramifications of these choices are discussed in Section V-B.

D. Hypotheses

Compile-time analysis is always limited by the amount of information that is available. The majority of static-analysis problems remain undecidable at compile time. The same applies to the polyhedron model. Therefore, we expect the model to be more useful when given more information about the program executed. This general expectation leads to the formalization of the hypotheses tested with our experiments:

H_1: $\textit{ExecCov}_{\text{Dyn}}$ is significantly greater than $\textit{ExecCov}_{\text{Stat}}$ ($\textit{ExecCov}_{\text{Dyn}} > \textit{ExecCov}_{\text{Stat}}$), on average.

H_1 follows naturally from the ordering of *ExecCov* given in Section III-B ($\textit{ExecCov}_{\text{Stat}} \leq \textit{ExecCov}_{\text{Dyn}}$). However, it is not clear whether applying polyhedral optimization at run time provides any benefit. It is necessary to verify that the run time spent in newly found SCoPs contributes significantly to the program's total run time.

H_2: The benefits of applying the polyhedron model just in time ($\textit{ExecCov}_{\text{Dyn}} - \textit{ExecCov}_{\text{Stat}}$) differ significantly across different domains (*Dom*), on average.

Synthetic benchmarks suggest that different application domains do not benefit equally from compile-time polyhedral optimization. H_2 makes the same statement for run-time polyhedral optimization (Class *Dynamic*).

The above two hypotheses are concerned with the issue of whether the differences of *Static* and *Dynamic* are statistically significant. Additionally we pose the following research question:

R_1: Is the difference between *Static* and *Dynamic* relevant in practice?

We consider an increase in execution coverage of around 10% or more practically relevant.

TABLE I: Description of dependent, independent, and controlled experimental variables. Dependent variables are influenced by independent and controlled variables. Independent variables are not influenced by other variables. Controlled variables influence dependent variables, but are fixed during the experiments.

Name	Abbreviation	Type	Scale type	Unit	Range
Execution SCoP coverage	*ExecCov*	Dependent	Ratio	%	$[0, 100]$
Application domain	*Dom*	Independent	Nominal	Text	{Compilation, Compression, Databases, Encryption, Multimedia, Scientific, Simulation, Verification}
Testing class	*Class*	Independent	Nominal	Text	{*Static*, *Dynamic*}
Test input	*Input*	Controlled	Nominal	Text	see Table II

E. Subject Programs

We conducted our study on the basis of an unbiased selection of open-source programs, as listed in Table II. Basically, we considered which domains are typically targets of a polyhedral optimization (Encryption, Multimedia, Scientific, Simulation) and selected at least two subject programs from each. Our intention for this selection was to choose programs that are well-known in the community. To get a broader picture, we selected subject programs of domains that are typically not targets of a polyhedral optimization (Compilation, Compression, Database, Verification) in a similar fashion. This selection was not entirely random: we were forced to choose programs that are compatible with our measurement infrastructure (see Section III-H).

F. Tasks

To set up a real-world scenario for each of the programs tested, we relied on benchmarks. We selected two different kinds of benchmarks depending on their availability. Our preferred test input was a benchmark that is commonly being used in the respective application domain, e.g., the reference input data for compression programs of SPEC2006. In the case of the absence of such a benchmark, we resorted to benchmarks used by the developers of the program. The assumption here was that a commonly accepted benchmark targets real-world scenarios more objectively than a benchmark used by the developers. The set of programs under investigation is distributed across a wide range of application domains. Table II shows a complete list of all programs tested, as well as their input parameters in our tests.

G. Design

Based on the variables introduced in Section III-C, we performed two experiments to validate our hypotheses and to answer our research question. In the first experiment, we compared *ExecCov* of the two classes *Static* and *Dynamic*. The second experiment compared the difference in *ExecCov* between *Static* and *Dynamic* across different domains.

Each subject program consists of two different program instances (one for each class) and belongs to one of eight application domains. Our experiments measure the execution SCoP coverage (*ExecCov*) for each program instance.

H. Experiment Setting

In our experiments, we used the Low-Level Virtual Machine (LLVM) compiler framework [9]. Its common representation during all stages of the compilation is the *LLVM Intermediate Representation* (LLVM-IR). LLVM-IR is a strongly typed, static single-assignment (SSA) language that connects to a wide variety of high-level languages. High-level languages are supported via different frontends, which provide LLVM-IR as output, e.g., CLANG [12] for C/C++, RUBINIUS [13] for Ruby, and PYPY [14] for Python. LLVM-IR can be optimized independently of platform and language. LLVM performs polyhedral optimization using the plugin POLLY [6], [7]. POLLY retrieves information about SCoPs from compatible loop nests found in the LLVM-IR. POLLY's SCoP detection

TABLE II: Subject programs and benchmarks used in the empirical study. A detailed description of test inputs can be found on the project's Web site (version numbers in parentheses indicate GIT commit hashes).

Name	Version	Tested inputs
Compilation		
SPIDERMONKEY	1.8.5	Integrated tests & SunSpider benchmark
PYTHON	3.2.3	Integrated tests
RUBY	1.9.3-p286	Integrated tests
SDCC	3.2.0	Integrated tests & Dhrystone benchmark
TINYCC	(9966fd4)	Integrated tests
Compression		
7Z	9.20.1	SPEC2006 input data (set: ref)
BZIP2	1.0.6	SPEC2006 input data (set: ref)
GZIP	1.2.4	SPEC2006 input data (set: ref)
XZ	5.1.1alpha	SPEC2006 input data (set: ref)
Database		
LEVELDB	(c8c5866)	Integrated benchmark
SQLITE3	3.7.13	Leveldb's integrated benchmark
POSTGRESQL	9.1.2	Integrated benchmark
Encryption		
OPENSSL	1.0.0e	Integrated test-binaries
CCRYPT	1.9	Integrated tests
LIBMCRYPT	2.6.8	Integrated benchmark
Multimedia		
LIBAV	(be64629)	Integrated benchmark
POVRAY	3.6.1	Integrated sample scenes
X264	(8a62835)	Integrated benchmark
Scientific		
LAMMPS	11/19/2011	Integrated sample problems
LAPACK	3.4.1	Integrated tests
LINPACK	2/25/94	Integrated benchmark
Simulation		
LULESH	1.0.1	Integrated benchmark
LULESH-OMP	1.0.1	Integrated benchmark
CRAFTY	20.0	Integrated benchmark
Verification		
CROCOPAT	2.1.5	Integrated benchmark
MINISAT	2.2.0	SAT-Race 2008 (reduced)

TABLE III: Preoptimization used for all LLVM-IR files throughout the empirical study.

Preoptimization pass	Purpose
-mem2reg	Promote memory to registers
-instcombine	Instruction combiner
-simplify-cfg	Clean up the CFG
-tailcallelim	Eliminate tail calls
-reassociate	Reassociate expressions
-loop-rotate	Rotate loops
-indvars	Simplify induction variables
-polly-region-simplify	Single-Entry-Single-Exit regions

is able to derive a polyhedral description of the loop nest if the restrictions of class *Static* are obeyed.

On top of POLLY, we built an open-source polyhedral analysis engine that is independent of the syntax of the source language, and that simulates all extensions necessary to detect SCoPs that are in class *Dynamic* by assuming that the necessary run-time information is present at the time of SCoP detection. This in itself is a valuable contribution to the community interested in experimenting with polyhedral optimization.

To assist POLLY in identifying SCoPs in LLVM-IR code, various preprocessing steps must be performed, as shown in Table III. These steps correspond to a relevant subset of the optimizations of LLVM's optimization switch O3 (We omitted irrelevant steps such as *jump-threading*). In our experiments, we use the same steps as POLLY.

To detect SCoPs in LLVM-IR, POLLY has to reconstruct all necessary information about loops, memory accesses and conditions. Looking at a program's control flow, a SCoP can be represented as a *region* in the *control-flow graph* (CFG). Assuming that no restriction of the polyhedron model is violated, a region in the CFG is a SCoP if it satisfies the *Single-Entry-Single-Exit* (SESE) property. LLVM includes an analysis pass to generate the required region information, based on a refined version of the program-structure tree [15]. We have created one further optimization step, *polly-region-simplify*, as shown in Table III, which establishes the SESE property for every region found in the CFG. A (simplified) schematic CFG of the example SCoP in Figure 1 is shown in Figure 3. The SCoP ranges across all basic blocks between `BB2` and `BB8` and has the SESE property. Due to this property, it is possible to introduce two additional basic blocks –`BB1` and `BB9`– into the CFG, which we use to place timing calls as close to the SCoP as possible.

We use instrumentation to retrieve precise timing information instead of sampling. Instead of deriving timing information for SCoPs based on common profiling frameworks such as OPROFILE [16] or GPROF [17], we have implemented the instrumentation based on the Performance Application Programming Interface (PAPI) [18]. It allowed us to retrieve timing information precisely at the entry and exit edges of a SCoP with high-precision timers. Among the four possible clock variants (real, virtual, user, system), we have chosen virtual time for our measurements. Virtual time consists of a process's user time (time spent in user mode) and system time (time spent in privileged mode). The ramifications of this choice are discussed in Section V-B.

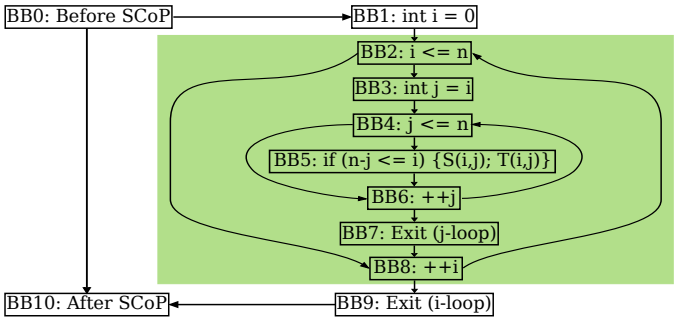

Fig. 3: Schematic view of the control-flow graph of Figure 1. The highlighted region marks the SCoP detected by POLLY.

Our measurements are integrated into the build system. To avoid repeated compilation of high-level source code, we stored each program in statically linked LLVM-IR. For future use, we decided to skip any preprocessing of these IR files and apply optimizations, as shown in Table III, before scanning for and instrumenting any SCoPs. This kind of instrumentation is bound to a SCoP. Therefore, its overhead depends on the number of SCoPs detected in the program. This number varies between the two classes. Thus, each sample program had to be run with two different binaries, one for each class, as introduced in Section III-G. With the instrumentation, we were able to calculate the necessary *ExecCov* for each program run.

In class *Static*, we have implemented SCoP detection by instrumenting all SCoPs detected by POLLY, without further modification of the detection process. We inserted the instrumentation only at transition edges between SCoPs. For class *Dynamic*, we had to track all failures during POLLY's SCoP detection. As SCoPs can be represented as nodes in the region tree, we analyzed the rejected SCoPs bottom-up and accepted those that are valid SCoPs under the assumption that sufficient run-time information is available (e.g., we would be able to insert parameter values into non-linear expressions induced by strong parametrization, as shown in Figure 2b).

If we were able to accept a SCoP with the assumed run-time information, we proceeded with POLLY's SCoP detection and expanded the detected region to the maximally possible size.

We conducted all experiments on an Intel i5 M520 machine (2 physical cores, 2.40 GHz) with 4GB RAM. To reduce fluctuations of our run-time measurements, we ensured (using the FIFO scheduling class in Linux) that the programs measured could not be preempted during execution.

I. Deviations

We experienced measurement bias in a few cases, caused by the introduction of one instrumented program binary per class. Since every instance required different SCoPs to be instrumented, we noticed different function-body alignment and cache effects, which led to minor inconsistencies in the results for *ExecCov* ($ExecCov_{\mathrm{Stat}} > ExecCov_{\mathrm{Dyn}}$). This circumstance does not affect our conclusions, as we discuss in Section V-B.

IV. Analysis

All results of our measurements are listed in Table IV (at the end of the paper). The complete set of experimental data is available at the project's Web site.

A. Descriptive Statistics

Figure 4 shows the distributions of execution coverage of the two classes *Static* and *Dynamic*; Table V lists the corresponding values of mean, variance, and standard deviation.

TABLE V: Mean (μ), standard deviation (s), variance (s^2) and median (m) for both classes.

Class	μ	s	s^2	m
Static	10	14	190	5.1
Dynamic	29	26	680	24

Looking at Table V, the execution coverage of *Static* is lower than that of *Dynamic*, but the variance of *Dynamic* is considerably higher than that of *Static*. We discuss this in Section V.

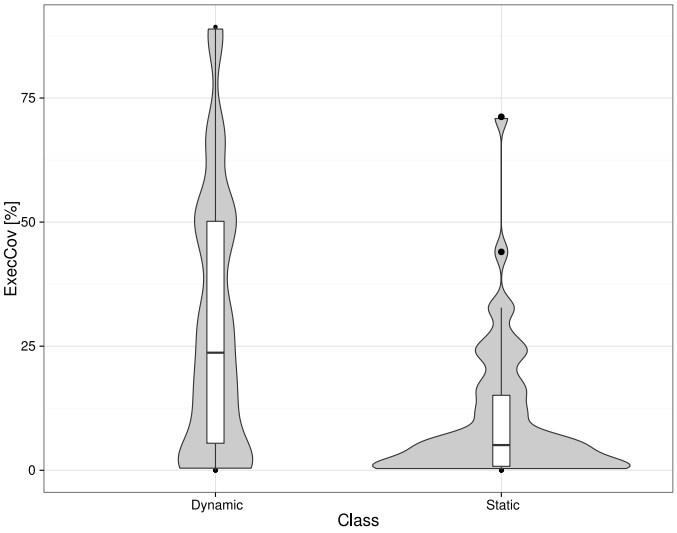

Fig. 4: Distributions of *ExecCov* for *Static* and *Dynamic* as violin plots [19] (combination of box plots and kernel density plots). The low end of the box represents the lower quartile (25 percentile), the top end the upper quartile (75 percentile). The bottom/top whiskers mark the lowest/highest datum in the 1.5 interquartile range of the lower/upper quartile. The band inside the box denotes the median. The violin shape around the box describes the estimated probability density of the data at different points. The grey violin shapes for *Static* and *Dynamic* cover the same area.

The violin shapes in Figure 4 describe the estimated probability density of the data at different execution coverages for *Static* and *Dynamic*. As a rule of thumb, the larger the area of the violin shape toward the top of the diagram (i.e., larger *ExecCov* values), the more we gain from polyhedral optimization. Inside the violin shape, there is a standard box plot, including median (horizontal line within the box), 25 and 75 percentile (bottom and top of the box).

Class *Static* has one large area of high density at the bottom, close to the median. This indicates that our subject set contains many programs with low *ExecCov* values for *Static*. Using run-time information in class *Dynamic*, the violin shape changes; the area at higher execution coverages become larger, indicating more programs with higher *ExecCov* values than with *Static*. Notice also the difference in the medians for *Static* and *Dynamic*. We discuss these distributions further in Section V. Table VI lists the average execution coverages as well as the relative differences between the two classes on a per-domain basis. An immediate observation is that the coverages and their relative differences vary considerably between individual domains, for example, Scientific ($+37\%$) compared to Database ($+3\%$), which we discuss in Section V.

TABLE VI: Mean (μ) dynamic coverage and difference between domains (in %): Δ denotes the benefit between the left and right column.

Domain	μ_{Stat}	Δ	μ_{Dyn}
Compilation	6.8	14	21
Compression	9.3	5.1	14
Database	9.6	3.2	13
Encryption	11	17	28
Multimedia	18	18	35
Scientific	4.2	37	41
Simulation	30	6.2	36
Verification	5.0	0.85	5.9

B. Hypothesis Testing

Let us comment on the significance of our measurements. Our case study consists of two independent data sets (*Class*). A Shapiro-Wilk Test [20] on the execution SCoP coverage (*ExecCov*) data of the classes *Static* ($p \ll 0.05$) and *Dynamic* ($p < 0.05$) reveals that none of the two data sets is distributed normally. So we use a Mann-Whitney-U Test [21].

Concerning H_1, a Mann-Whitney-U Test reveals that $\mathit{ExecCov}_{\mathrm{Dyn}}$ is significantly greater than $\mathit{ExecCov}_{\mathrm{Stat}}$ ($p \ll 0.05$).

Concerning H_2, a Kruskal-Wallis Test [22] reveals that the benefit of *Dynamic* ($\mathit{ExecCov}_{\mathrm{Dyn}} - \mathit{ExecCov}_{\mathrm{Stat}}$) differs significantly across different domains ($p \ll 0.05$). This is also shown in Table VI (e.g., consider the average benefit of Scientific ($37\,\%$) vs. the average benefit of Verification ($0.85\,\%$)). In summary, we can accept both of our hypotheses.

With regard to our research question R_1, we found that the increase in execution coverage induced by using run-time information is not only statistically significant but also practically relevant. Across all domains, the execution coverage of class *Dynamic* is 19% higher, on average, up to 37% for particular domains, than the execution coverage of class *Static*.

V. Discussion

Next, we discuss the consequences of our results for polyhedral analysis and optimization in detail.

A. Results

We begin with the potential of compile-time polyhedral analysis (class *Static*). In our experiments, polyhedral analysis at compile time is able to optimize 0.79–15.1 % (as shown

in Figure 4) of a program's total run time. Exceptions are SHA512, LULESH-OMP (the two shapes at the top of *Static*'s violin plot in Figure 4). SHA512 is a very short-running benchmark ($\mathbf{T}_{\mathrm{Stat}} = 0.024s$), which increases the influence of a detected SCoP; LULESH-OMP is hand-optimized code for OPENMP parallelization, which increases the chances of compatible loops due to the regular nature of OPENMP codes; only POSTGRES showed unexpected results: we did not expect a database management system to achieve such a high coverage ($\mathit{ExecCov}_{\mathrm{Stat}} = 24\%$). While these examples illustrate that compile-time polyhedral optimization may be selectively highly beneficial, the overall picture is that, in many cases, it could not play to its strengths and is practically limited.

As soon as run-time information is available to the polyhedral analysis (class *Dynamic*), *ExecCov* rises to 5–50 % (as shown in Figure 4); this increase is significant (H_1).

A notable observation is that the descriptive variance ($s^2 = 680$) of $\mathit{ExecCov}_{\mathrm{Dyn}}$ is significantly greater ($p \ll 0.05$) than $\mathit{ExecCov}_{\mathrm{Stat}}$ ($s^2 = 190$). According to hypothesis H_2, $\mathit{ExecCov}_{\mathrm{Dyn}}$ differs significantly from $\mathit{ExecCov}_{\mathrm{Stat}}$ across different domains. This suggests that the high variance is caused by a considerable difference between individual domains. Table VI reveals that the domains Multimedia (+18 %) and Scientific (+37 %) gain the most when applying the polyhedron model at run time, confirming the common belief that these domains are well-suited for polyhedral optimization. The smallest benefit was achieved in the domains Verification (+0.85 %) and Database (+3.2 %). This complies with the common expectation that these domains are not well-suited for polyhedral optimization.

So, our study demonstrates that the use of run-time information in polyhedral optimizations is of practical relevance.

B. Threats to Validity

Construct validity: Our measurement method is based on different instrumentations per binary per class. This circumstance has an influence on the run-time fraction of SCoPs measured. The differing instrumentations are necessary because we have to detect the SCoPs based on the testing class and instrument each detected SCoP. For each program of each class, we compared the difference between instrumented and uninstrumented run time to the expected overhead caused by our instrumentation. We found that our instrumentation does not have a negative influence on the run time of any individual SCoP in the program.

As mentioned in Section III-H, we measured virtual time with high-precision timers provided by the operating system. It is obvious that instrumentation causes a higher run-time overhead than sampling. Mainly, this overhead is generated due to the instrumented calls necessary to obtain timing information. However, it is possible that the instrumented code suffers from other negative side effects, e.g., cache effects or ineffective function body alignment. Note that our instrumentation is bound to the entry and exit of a SCoP. Thus, it could possibly slow down a SCoP more than other program parts, which would increase *ExecCov*. We verified that this did not cause problems during our experiments by comparing the calibrated average run-time (*CART*) of one instrumentation call with the total run-time difference between the instrumented program and the uninstrumented program divided by the number of instrumentation calls executed (actual average run time of one instrumentation call (*RART*)). *CART* and *RART* are of the same order of magnitude, so significant slow-downs caused by our instrumentation are unlikely.

Internal validity: Our experiments relied on the quality of our input data, because we measured run time. We have addressed this threat to validity by choosing the developer's own benchmark sets or by using the known default benchmark of the according domain, such as SUNSPIDER [23] for a JavaScript engine. This does not remove the dependence on *Input*, but we assume that the developers' own test cases cover the important code paths and the default benchmarks of a domain cover the most common use cases.

External validity: As with any other comparable study, the selection of sample systems threatens the generalizabilty of our results. We controlled this threat reasonably by selecting a large and diverse number of subject systems randomly.

C. Perspectives

In our experiments, polyhedral optimization does not show great potential when applied at compile time, but this does not necessarily imply that the polyhedron model, in general, is not suited for application at compile time. The tools we use for determining the coverage data (LLVM, POLLY) are practical tools that implement only a subset of the polyhedron model yet. Extensions proposed in academia may flow into these tools and improve the situation (e.g., correct handling of integer wrapping and multi-dimensional arrays in LLVM).

Having said this, our study demonstrates for the first time the potential of applying polyhedral optimizations at run time. Using run-time information increased the dynamic coverage substantially (up to 29% on average). We consider this observation a major result and an encouragement for the community to follow this path, both in terms of refining and extending the model and by extending practical tools accordingly.

Giving up the restriction to affine linearity would unfold the full potential of polyhedral optimization that is theoretically possible and in reach of the polyhedron model. Therefore, it is interesting to quantify the extensibility of the polyhedron model with respect to execution SCoP coverage. In a series of further experiments, we considered an additional SCoP class, which only required static knowledge of the control flow, called *Extended*, to investigate all possibilities of applying polyhedral optimization at both compile time and run time.

The experiments and a detailed discussion of class *Extended* can be found elsewhere [24]. In a nutshell, we found that *Extended* increases the benefit of polyhedral optimization significantly ($p < 0.05$) further by another 10%, on average. Although this average increase seems low, it is important to note that it is significantly ($p \ll 0.05$) higher in domains that did not benefit from run-time information available in class *Dynamic*. This shows that there is still a lot of room for techniques

that extend the polyhedron model beyond linear affinity, for example, dealing with polynomial loop bounds [5].

VI. Related Work

Let us review other studies, extensions of the polyhedron model, and alternative technologies related to our work. The extensions we discuss here have the potential to enable access to SCoPs of classes *Dynamic* and *Extended*.

A. Alternative Studies

Alnaeli et. al. [25] conducted an empirical study on the parallelizability of open-source systems. They studied the evolution of parallelization opportunities for 11 open-source software systems. They conclude that the main problem with the programs tested are function calls inside the loop bodies. Therefore, future research should focus more on dealing with side effects in function calls. In contrast to our work, they did not investigate the run-time fraction of the parallel loops found.

B. Alternative Extensions

The following extensions focus on transcending affine linearity at compile time.

Benabderrahmane et al. [26] model arbitrary, non-recursive, control flow within a SCoP at compile time, converting control dependences to data dependences if necessary. The same approach can be used to deal with **while** loops in SCoPs. A **while** loop is transformed to an unbounded **for** loop, and an exit conditional is introduced in the body in form of a write access [27]. Every existing statement depends on this exit conditional, thus terminating the loop execution if the condition is violated. These capabilities come at the cost of a loss of precision of the whole analysis. In particular, the dependence introduced to the exit conditional forces the scheduler to generate a sequential schedule.

In contrast to stretching the modeling capabilities by giving up precision, there are a few extensions to the polyhedron model that cope with non-linearity by using new algebraic methods, without giving up precision. First, it is possible to deal with multiplicative parameters throughout modeling, transformation and code generation at compile time by using real quantifier elimination [5]. Second, cylindrical algebraic decomposition can be used to provide support for input programs that feature more complicated non-linearity, such as polynomials in the index variables [5]. However, both approaches suffer from significant performance penalties during code synthesis as well as in the generated code itself.

C. Alternative Technologies

Beside the LLVM framework, there are several other compiler frameworks that support the polyhedron model.

Most earlier and some current systems extract SCoPs directly from the program source code. This requires a syntactic markup of SCoPs. LooPo [28] was the first such system. A recent system is PoCC [29], which implements a full compiler tool chain for automatic polyhedral optimization. It supports two polyhedral transformation tools: Pluto [30], [31] and LetSeE [32], [33]. The Pluto scheduling algorithm implements a transformation that optimizes data locality on shared-memory systems. Rather than generating the optimal solution, LetSeE tries to converge on it iteratively by exploring the legal transformation space.

Recent implementations of the polyhedron model work on a compiler's intermediate representation (IR), e.g., Polly. The main advantage is that, unlike with tools that work on source code, SCoPs need not be written in a fixed syntactic form, since the loop structure and array accesses are obtained from a loop and pointer analysis on the IR. A project similar to Polly is GCC's Graphite [8]. Other implementations working on the IR include Wrap-IT [34] (based on Open64) and the IBM XL compiler [35].

Research on the field of run-time polyhedral optimization has started to emerge. The latest contribution by Jimborean [36] merges speculative techniques, such as the LRPD test [37], and adaptive compilation with polyhedral optimization. However, most of the polyhedral optimization is still performed at compile time.

VII. Conclusions and Future Work

The polyhedron model is a well-studied and promising approach to automatic program optimization. By means of an empirical study of the potential of polyhedral optimization –the first study of its kind– we have demonstrated that current practical implementations of the polyhedron model do not achieve practically relevant execution coverages, when applied to real-world programs at compile time (10 %). However, we found that a polyhedral analysis benefits significantly from the available amount of information when applied at run time (the code regions that can be covered increase from 10 % to 29 % on average). Our study suggests that the time is ripe for researchers and tool builders to tap into this potential. Whether it will be sufficient to outweigh the run-time overhead spent on performing the optimizations at run time must be answered by future studies. Furthermore, overcoming the limits of affine linearity can increase the dynamic coverage, which encourages to push the boundaries of the polyhedron model further toward practical application.

Beside the material and results of our empirical study, we contribute our polyhedral analysis engine to help other researchers to conduct empirical studies on polyhedral optimization. Our set of subject programs and benchmarks is a good start for a community effort to coordinate work on improving polyhedral optimization.

Acknowledgements

We thank Christian Kästner and Albert Cohen for comments on earlier drafts of this paper. This work was partially supported by the DFG projects PolyJIT (grant no. GR 4253/1 and LE 912/14) and ExaStencils (grant no. AP 206/7 and LE 912/15).

TABLE IV: Execution SCoP coverage of the analysis classes *Static* and *Dynamic* (in %). Values for class *Extended* are shown on the rightmost side (see Section V-C). The column $\mathbf{T}_{\text{Raw}}$ shows the run time of the program without instrumentation. The columns $\mathbf{T}_{\text{Stat}}$,$\mathbf{T}_{\text{Dyn}}$, $\mathbf{T}_{\text{Ext}}$ show the run time in the respective class (in s).

Name	Stat	$\mathbf{T}_{\text{Stat}}$	Dyn	$\mathbf{T}_{\text{Dyn}}$	$\mathbf{T}_{\text{Raw}}$	Stat | Dyn	Ext	$\mathbf{T}_{\text{Ext}}$
Compilation								
PYTHON	0	0	0	0	0		0	0
RUBY	27	770	68	450	330		75	1500
SDCC	4.6	37	13	45	34		27	67
JS	0.31	6.1	21	11	5.9		38	26
TCC	1.7	1.0	1.9	1.0	0.78		19	1.6
Compression								
7ZA	17	88	17	87	58		36	180
BZIP2	5.1	18	18	23	16		48	150
GZIP	2.6	12	3.8	12	13		36	38
XZ	12	170	19	200	120		38	440
Database								
LEVELDB	1.4	0.058	1.7	0.077	59		0	0.076
POSTGRES	24	270	26	290	0		45	420
SQLITE3	3.4	210	11	250	130		43	1200
Encryption								
CCRYPT	0.34	0.12	14	0.16	0		45	0.94
MCRYPT-AES	0.99	0.000 61	1.0	0.000 77	0		4.8	0.0010
MCRYPT-CIPHERS	12	0.052	13	0.054	0.020		38	0.14
BLOWFISH	0.	0.0030	0.85	0.0031	0		0.94	0.0031
BN	17	1.4	42	2.7	2.0		50	13
CAST	23	1.8	24	1.8	2.2		52	2.6
DES	1.1	0.000 37	1.1	0.000 35	0		36	0.0013
DSA	15	0.066	32	0.13	0.12		45	0.41
ECDSA	0.60	1.5	30	3.7	3		49	35
HMAC	5.6	0.000 25	11	0.000 11	0		12	0.000 14
MD5	1.3	0.000 38	5.7	0.000 14	0		4.2	0.000 14
RC4	0	6.4×10^{-5}	0	6.6×10^{-5}	0		0	6.4×10^{-5}
RSA	25	0.98	28	1.1	1.2		45	5.4
SHA1	0.066	0.0046	89	0.0048	0		88	0.0049
SHA256	0.042	0.024	70	0.036	0.020		71	0.036
SHA512	71	0.023	88	0.028	0.010		88	0.030
Multimedia								
AVCONV	33	390	46	950	160		50	2200
POVRAY	15	450	28	630	310		90	780
X264	4.6	25	33	55	22		40	81
Scientific								
XEIGTSTC	6.6	16	52	54	14		53	65
XEIGTSTD	9.6	15	50	61	12		51	74
XEIGTSTS	11	13	52	58	10		52	69
XEIGTSTZ	4.8	20	51	61	18		51	73
XLINTSTC	0	0	0	0	15		0	0
XLINTSTD	0	0	0	0	8.5		0	0
XLINTSTDS	5.4	2.3	64	9.3	2.3		65	9.8
XLINTSTRFC	2.1	6.7	50	19	6		51	21
XLINTSTRFD	6.9	2.8	55	15	2.5		55	16
XLINTSTRFS	6.0	2.8	55	15	2.5		55	16
XLINTSTS	0	0	0	0	7.9		0	0
XLINTSTZ	0	0	0	0	17		0	0
XLINTSTZC	0.45	3.5	65	8.7	3.4		66	8.9
LINPACK	5.8	16	85	17	34		85	17
Simulation								
CRAFTY	33	150	36	180	50		49	760
LAMMPS	24	570	42	1600	300		50	2700
LULESH	20	360	21	360	220		34	560
LULESH-OMP	44	1300	47	1700	280		48	1900
Verification								
CROCOPAT	3.5	230	5.2	230	150		20	340
MINISAT	6.6	2700	6.6	2700	2300		10.0	2900

REFERENCES

[1] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1581–1592.

[2] L. Rauchwerger, N. M. Amato, and D. A. Padua, "A scalable method for run-time loop parallelization," *Int'l J. Parallel Programming*, vol. 23, no. 6, pp. 537–576, 1995.

[3] R. Asenjo, R. Castillo, F. Corbera, A. G. Navarro, A. Tineo, and E. L. Zapata, "Parallelizing irregular C codes assisted by interprocedural shape analysis," in *Proc. Int'l Symp. Parallel and Distributed Processing (IPDPS)*. IEEE CS, 2008, pp. 1–12.

[4] J.-F. Collard, "Automatic parallelization of *while*-loops using speculative execution," *Int'l J. Parallel Programming*, vol. 23, no. 2, pp. 191–219, 1995.

[5] A. Größlinger, "The challenges of non-linear parameters and variables in automatic loop parallelisation," Doctoral thesis, Department of Informatics and Mathematics, University of Passau, 2009.

[6] T. Grosser, H. Zheng, R. Alor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly – polyhedral optimization in LLVM," in *Proc. Int'l Workshop Polyhedral Compilation Techniques (IMPACT)*, 2011.

[7] T. Grosser, A. Größlinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012, 28 pp.

[8] 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. Int'l Workshop GCC Research Opportunities (GROW)*, 2010, pp. 1–13, http://ctuning.org/workshop-grow10.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int'l Symp. Code Generation and Optimization (CGO)*. IEEE CS, 2004, pp. 75–86.

[10] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*. IEEE CS, 2004, pp. 7–16.

[11] M. Davis, "Hilbert's tenth problem is unsolvable," *American Mathematical Monthly*, vol. 80, pp. 233–269, 1973.

[12] "Clang: A C language family frontend for LLVM," http://clang.llvm.org.

[13] "Rubinius: An environment for the Ruby programming language," http://rubini.us.

[14] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo, "Allocation removal by partial evaluation in a tracing JIT," in *Proc. Int'l Workshop Partial Evaluation and Program Manipulation (PEPM)*. ACM, 2011, pp. 43–52.

[15] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proc. Int'l Conf. Programming Language Design and Implementation (PLDI)*. ACM, 1994, pp. 171–185.

[16] J. Levon, "OProfile – a system profiler for linux," http://oprofile.sourceforge.net/doc/, 2007.

[17] J. Fenlason and R. Stallman, "GNU gprof," http://www.gnu.org/manual/gprof-2.9, 1988.

[18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int'l J. High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[19] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.

[20] S. Shapiro and M. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[21] H. Mann and D. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[22] W. Kruskal and W. Wallis, "Use of ranks in one-criterion variance analysis," *J. of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[23] "SunSpider: JavaScript Benchmark," http://www.webkit.org/perf/sunspider/sunspider.html, 2012.

[24] A. Simbürger, S. Apel, A. Größlinger, and C. Lengauer, "The potential of polyhedral optimization," University of Passau, Tech. Rep. MIP-1301, 2013.

[25] S. M. Alnaeli, A. Alali, and J. I. Maletic, "Empirically examining the parallelizability of open source software systems," in *Proc. Int'l Conf. Working Conference Reverse Engineering (WCRE)*. IEEE CS, 2012, pp. 377–386.

[26] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proc. Int'l Conf. Compiler Construction (CC)*, ser. LNCS, vol. 6011. Springer, 2010, pp. 283–303.

[27] M. Griebl and C. Lengauer, "On the space-time mapping of while-loops," *Parallel Processing Letters*, vol. 4, no. 3, pp. 221–232, 1994.

[28] ——, "The loop parallelizer LooPo—Announcement," in *Proc. Int'l Workshop Languages and Compilers for Parallel Computing (LCPC)*, ser. LNCS. Springer, 1997, vol. 1239, pp. 603–604.

[29] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *Proc. Int'l Conf. High Performance Computing Networking, Storage and Analysis (SC)*. IEEE CS, 2010, pp. 1–11.

[30] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. Int'l Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 101–113.

[31] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *Proc. Int'l Conf. Compiler Construction (CC)*, ser. LNCS, vol. 4959. Springer, 2008, pp. 132–146.

[32] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative optimization in the polyhedral model: Part I, one-dimensional time," in *Proc. Int'l Symp. Code Generation and Optimization (CGO)*. IEEE CS, 2007, pp. 144–156.

[33] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: Part II, multidimensional time," in *Proc. Int'l Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2008, pp. 90–100.

[34] 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'l J. Parallel Programming*, vol. 34, pp. 261–317, 2006.

[35] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques (PACT)*. ACM, 2010, pp. 343–352.

[36] A. Jimborean, "Adapting the polytope model for dynamic and speculative parallelization," Doctoral thesis, Image Sciences, Computer Sciences and Remote Sensing Laboratory, University of Strasbourg, 2012.

[37] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 2, pp. 160–180, 1999.