The Potential of Polyhedral Optimization

Andreas Simbürger, Sven Apel, Armin Größlinger, Christian Lengauer

Department of Informatics and Mathematics, University of Passau
{simbuerg,apel,groessli,lengauer}@fim.uni-passau.de

Technical Report, Number MIP-1301
Department of Informatics and Mathematics
University of Passau, Germany
February 2013

The Potential of Polyhedral Optimization

Andreas Simbürger Sven Apel Armin Größlinger Christian Lengauer
University of Passau
{simbuerg,apel,groessli,lengauer}@fim.uni-passau.de

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 optimization 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 optimization shows greater relevance for real-world applications. On average, the share of the run time amenable to polyhedral optimization is increased by a factor of nearly 3.

Categories and Subject Descriptors D.2.8 [*Metrics*]: Performance Measures; D.3.4 [*Processors*]: Optimization; F.1.2 [*Modes of Computation*]: Parallelism and Concurrency

General Terms Experimentation, Languages, Measurement, Performance

Keywords Polyhedron Model, JIT, Loop Parallelisation, LLVM, Polly

1. 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 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 [5, 13, 14, 32]. One popular approach is *polyhedral optimization*. It is based on the *polyhedron model* [14], which represents iterative executions as polyhedra. The model serves to optimize loop programs automatically (e.g., parallelization and data localization) by applying algebraic transformations. Its main benefit is that all loop transformations can be found via linear optimization and, therefore, their cost 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 accesses are limited to affine linear expressions. In the past, a number of extensions of the polyhedron model have been proposed to overcome affine linearity in certain cases [20]. 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 [18, 19] and GRAPHITE [34]. 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 potential benefits of the model and its variants in practice across different domains. Therefore, for the first time, we conducted a comprehensive empirical study on the potential benefit of polyhedral optimization for automatic loop parallelization in practice. In particular, (1) we were interested in whether state-of-the-art polyhedral methods are applicable in practice; (2) we were interested in how far this potential can be enhanced by an application at run time; (3) we were interested in the upper bound of the benefit of the polyhedron model and extensions thereof.

In an empirical study, we have analyzed a corpus of 51 real-world C/C++ programs, using (an extension of) the plugin POLLY for the LLVM [25] compiler infrastructure, including programs 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 is a more general approach than just quantifying the effects of a certain specific instance of polyhedral optimization.

We found that the potential benefit of state-of-the-art compile-time polyhedral optimization is rather marginal (10 % on average). This is mainly due to a lack of knowledge about parameter values at compile time, which introduces non-linearity of the memory accesses of a loop program. The potential benefit of run-time optimizations is higher (29 % on average). Just by supplying the missing information via a run-time program analysis, the SCoP coverage could be raised in cases from 4 % to 32 %.

Based on these findings and a calculation of the theoretical upper bound, 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 execution. However, by exploiting information available just-in-time, the model is capable of covering most of the critical program regions and providing program transformations to exploit the potential for parallelism.

Based on our findings, future research shall 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 discussion of the practical potential of compile-time and run-time polyhedral optimization based on the empirical results and a statistical analysis.

The analysis engine, the sample systems, 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/`

2. The Polyhedron Model

2.1 Basic Model

Let us introduce the necessary basics of the polyhedron model. A more comprehensive introduction can be found elsewhere [14]. The model serves to parallelize loop programs automatically by applying algebraic transformations. Its main benefit is that loop transformations can be derived fully automatically and their cost is independent of the problem size.

The polyhedron model provides an abstract representation for loop programs. 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).

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 $\vec{i}$. A *schedule* Θ_S for statement S is a function $\theta_S : D_S \to \mathbb{Z}^m$ that maps each instance of statement S to a single point in (multi-dimensional) time[1]; hence, the schedule determines the execution order of the statement instances via the lexicographic order in $\mathbb{Z}^m$. Each statement can contain memory *accesses* to arrays (the only aggregate data structure allowed in the model); these are represented by relations between the domain of the statement and the indices accessed of the respective array, e.g., $\{(S; \vec{i}) \mapsto (A; 2 \cdot \vec{i})\}$ for an access `A[2*i]` in statement S.

The most important computational task when using the polyhedron model for program transformations is to compute the data dependences between statement instances. A statement instance $(T; \vec{j})$ depends on an instance $(S; \vec{i})$ iff there exist memory accesses in S and T such that both statements access the same memory cell (for the given values of $\vec{i}$ and $\vec{j}$). Transformations of the program must not violate the dependences (by executing dependent instances in the wrong order or in parallel). To make the determination of the data dependences computable, the following restriction is imposed. The constraints of all domains, schedules, and memory access relations must be affine, i.e., all constraints can be written in the form $M\vec{i} \geq \vec{b}$ for $M \in \mathbb{Z}^{k \times n}$, $\vec{b} \in \mathbb{Z}^k$. Geometrically, these objects are ($\mathbb{Z}$-)polyhedra; hence, the name of the model. Note that some dimensions of $\vec{i}$ can be structure parameters, which allows parameters to occur additively (*weak* parametrization, e.g., Figure 2(a)) in all constraints, schedules and memory accesses, but not multiplicatively (*strong* parametrization, e.g., Figure 2(b)). This restriction implies that the control flow of the loop program (represented by the domains and schedules) must be known at compile time, i.e., the loop bounds and conditionals must be affine expressions. Thus, non-affine conditions and recursive control flow cannot benefit from polyhedral optimization.

[1] Note that schedule and iteration domain do not necessarily have an equal dimensionality.

```
1   int i,j;
2   for (i=0; i<=n; ++i)
3     for (j=i; j<=n; ++j)
4       if (i >= n-j) {
5 S:      A[i+n][j+i] = B[n+2*i-1][j];
6 T:      B[i+n][j-i] = A[n-2*i+1][j];
7       }
```

Figure 1. A static control program (SCoP).

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 only statement type allowed in the loop body is the assignment. Calls to 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 called function.

These strict requirements limit the number of programs that can be analyzed. Each loop in the program complying with them is called a *static control part* (SCoP) [6].

An example of a SCoP is given in Figure 1. The SCoP consists of two statements, labeled S and T. They share the domain $\{(i,j) \mid 0 \leq i \leq n \wedge i \leq j \leq n \wedge i \geq n - j\}$. Note that the condition $i \geq n - j$, derived from the conditional statement, is added to the constraints of the domain derived from the loop bounds. The control flow is known at compile time because the predicate and the loop bounds are affine expressions. 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:

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

The program in Figure 1 is sequential. Therefore, the schedule for its statements can be given by $\theta_S(i,j) = (i,j,0)$ and $\theta_T = (i,j,1)$. Notice that the lexicographic order in the range of the schedule yields the sequential execution order of the loop iterations (in the first two dimensions) and the textual order between statement instances of the same iteration (in the third dimension). The aim of polyhedral optimization is to find "better" schedules, e.g., schedules that speed up the execution via parallelism or that improve cache efficiency. The search for better schedules is not discussed in this paper; we are interested in the more general question of what fraction of real-world codes can be modeled in the basic model and in extensions thereof to be accessible by polyhedral optimization in the first place.

2.2 Classification of SCoPs

Successful detection of SCoPs in source code depends on two factors. First, the time of detection; is detection performed at compile time or at run time? Second, the extensions applied to the model to overcome certain restrictions, e.g., extensions to deal with multiplicative parameters via quantifier elimination in the reals [20].

Let us introduce three different classes of SCoPs – *Static*, *Dynamic* and *Extended* – which will be used throughout our empirical study. Each class defines a larger set of SCoPs than the previous one. It has to be noted that none of the run-time–based classes have been implemented in a production compiler yet. This made it necessary for us to extend LLVM to be able to detect SCoPs of classes *Dynamic* and *Extended*.

Class *Static* reflects the current state of the art of polyhedral optimization. Class *Dynamic* is one of the hot topics in the polyhedron research community. Class *Extended* can be viewed as an upper bound on practicality.

2.2.1 Class *Static*

This smallest class covers classical SCoPs, i.e., all SCoPs that can be represented in the basic polyhedron model and apply all analysis and transformation steps at compile time. Current implementations of the model (e.g., POLLY or GRAPHITE) are capable of detecting SCoPs of this class.

2.2.2 Class *Dynamic*

Much like class *Static*, class *Dynamic* incorporates also only SCoPs that can be represented in the basic polyhedron model. However, the SCoP detection of this class takes place at run time of the program to be analyzed.

At compile time, any possible violation of the model's restrictions in a given part of the code precludes polyhedral optimization. But, the knowledge available at run time enables a more precise evaluation of the adherence of 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.

Next, we describe three patterns that can be detected in class *Dynamic* in addition to class *Static*. We use these patterns also in our empirical study. All patterns assume that SCoP detection is performed at run time.

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

Figure 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.

In Figure 2(b), 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. In contrast to more sophisticated methods of dealing with non-linearity [20], this does not require any changes to the model.

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 transformed loop program 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.

Known Aliasing As soon as we consider input languages that support pointers, we have to deal with the possibility of aliasing. Present alias analyses can provide only a conservative approximation, leading to so-called *may-alias*, i.e., alias whose existence must be assumed but is uncertain. There are two alternative ways of dealing with a may-alias: (1) one installs 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 real aliasing is revealed at run time anyway and, therefore, we do not reject SCoPs that include unknown aliasing behavior.

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 the affine linear form $Ax \geq b$, and any side-effects of function calls must be known. Using 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 to functions whose entire bodies form valid SCoPs at compile time, i.e., the function body itself 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 which 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, hence, calling the function with an iterator as argument leads to non-linearities. However, we can permit functions that form valid SCoPs in class *Static* as the function arguments can only occur as linear parameters in the contained SCoP. We restrict the arguments at the function call to affine linear expressions. This restriction is conservative, as the uses of a called function's parameters determine the compatibility with the model as well. This leaves room for improvement in the class *Dynamic*.

Aside. The more practical alternative is to exploit run-time knowledge on program input values and parameters, and inline the function body just-in-time (excluding recursion). However, this technique is not restricted to the class *Dynamic* alone. Therefore, we exclude its investigation from this study.

2.2.3 Class *Extended*

As stressed previously, affine linearity is the key limiting factor in applying the polyhedron model at compile time or at run time. Some violations of this restriction at compile time dissolve when run-time information turns a variable into a constant (class *Dynamic*). For violations that remain, the restriction to affine linearity is removed in class *Extended*; only static knowledge of the control flow is required.

Conceptually, dropping the restriction to affine linearity opens the way for a multitude of aggregate data structures like lists, trees, etc. Obviously, many –especially irregular– access patterns possible on such structures will not be treatable by polyhedral methods. However, some have already shown to be treatable, e.g., polynomial loop bounds [20], and others will likely follow.

For now, we view class *Extended* more as an upper bound on what is possible, than as a terrain of practicality.

3. Experiment Planning & Execution

We conducted an empirical study to estimate the potential of polyhedral optimization of real-world programs. We are interested in the impact of polyhedral methods in three areas:

- application of the basic model at compile time,
- application of the basic model at run time,
- any conceivable relaxations of the basic model (upper bound).

3.1 Goals

Our empirical study has been designed to answer the following questions. What is the potential of the basic polyhedron model at compile time? Can the potential benefit of polyhedral methods be improved by using run-time information? Can the model's coverage be improved significantly by giving up the restriction to affine linearity, even if the result may not not be well-suited for optimization? How applicable is the polyhedron model augmented by run-time knowledge? How applicable is the polyhedron model without restrictions to affinity? Are there differences with regard to the above questions between different application domains?

3.2 Measurement Methodology

Empirical evaluation in the context of polyhedral optimization proceeds usually by measuring the effect of a specific instance of polyhedral optimization on the run time of benchmarks. However, 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 so-called *dynamic SCoP coverage*.

Definition (Dynamic 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.* Dynamic SCoP coverage *(DynCov) is then defined as:*

$$\mathit{DynCov} := (\Sigma s : s \in \mathbb{S} : t(s)) * \frac{1}{T}$$

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

Our experiments measure the dynamic SCoP coverage of each program in the three classes *Static* ($\mathit{DynCov}_{\mathrm{Stat}}$), *Dynamic* ($\mathit{DynCov}_{\mathrm{Dyn}}$) and *Extended* ($\mathit{DynCov}_{\mathrm{Ext}}$). Note that, even though *DynCov* can be ordered by $\mathit{DynCov}_{\mathrm{Stat}} \leq \mathit{DynCov}_{\mathrm{Dyn}} \leq \mathit{DynCov}_{\mathrm{Ext}}$, this does *not* imply *Static* $\subseteq$ *Dynamic* $\subseteq$ *Extended*. Every SCoP detected is maximized in size and, therefore, does not necessarily represent the same SCoP in a different (lower) class.

3.3 Experimental Variables

The variables of our empirical study are listed in Table 1. Our main focus lies on a single dependent variable, *DynCov*. It is influenced by the total run time and the input data of the executed program. The input data determine the code paths chosen in the program run and, therefore, controls the fraction of SCoPs that are executed during one program run. We chose to control the test input by making it constant per program (*Input*). However, we tracked the achieved code coverage of each program to ensure that the major fraction of code paths in the program have been visited at least once.

Additionally, *DynCov* depends on the program's total run time by definition. As the run time of a program itself depends on the input values, we do not consider the total run time as a variable for our experiments. However, the selected values of *Input* control the impact of the program's run time by assuming that the chosen benchmarks give an accurate view on the real-world execution of the tested program. Further ramifications of this exclusion are discussed in Section 5.2.

Our study requires two more variables to refine our findings, both of them independent. We distributed all tested programs across a wide variety of application domains (*Dom*). All programs that were expected to behave similarly or perform a similar task (e.g., compilation) were grouped in the same domain. Based on the introduction in Section 2.1, we collected *DynCov* for each program instance in each class (*Class*).

3.4 Hypotheses

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

$H_{1.1}$: $\mathit{DynCov}_{\mathrm{Dyn}}$ is significantly greater than $\mathit{DynCov}_{\mathrm{Stat}}$ ($\mathit{DynCov}_{\mathrm{Dyn}} > \mathit{DynCov}_{\mathrm{Stat}}$), on average.

$H_{1.2}$: $\mathit{DynCov}_{\mathrm{Ext}}$ is significantly greater than $\mathit{DynCov}_{\mathrm{Dyn}}$ ($\mathit{DynCov}_{\mathrm{Ext}} > \mathit{DynCov}_{\mathrm{Dyn}}$) and therefore transitively greater than $\mathit{DynCov}_{\mathrm{Stat}}$, on average.

$H_{2.1}$: The benefits of applying the polyhedron model just-in-time ($\mathit{DynCov}_{\mathrm{Dyn}} - \mathit{DynCov}_{\mathrm{Stat}}$) differ significantly across different domains (*Dom*), on average.

$H_{2.2}$: The benefits of extending the model beyond affine linearity ($\mathit{DynCov}_{\mathrm{Ext}} - \mathit{DynCov}_{\mathrm{Dyn}}$) differ significantly across different domains (*Dom*), on average.

3.5 Sample Programs

We conducted the study on an unbiased selection of open-source programs, as listed in Table 2. The selection was not entirely random, as we could select only programs that were compatible with our measurement infrastructure (see Section 3.8).

3.6 Tasks

In order to set up a real-world scenario for each of the tested programs, we relied on benchmarks. We selected two different kinds of benchmarks depending on their availability. Our preferred test input was a benchmark, which 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 2 shows a complete list of all programs tested, as well as their input parameters in our tests.

3.7 Design

Based on the variables introduced in Section 3.3, we performed two experiments to validate our hypotheses. In the first experiment, we compared *DynCov* of the three classes *Static*, *Dynamic* and *Extended*. The second experiment compared the difference in *DynCov* between classes across different domains.

Each sample program consists of three different program instances (one for each class) and belongs to one of eight application

Name	Abbreviation	Type	Scale type	Unit	Range
Dynamic SCoP coverage	*DynCov*	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*, *Extended*}
Test input	*Input*	Controlled	Nominal	Text	See Table 2

Table 1. 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 the dependent variables, but are fixed during the experiments

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 (1 sample video)
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 2. Projects from which we generated the sample programs for our experiments and benchmarks used in the empirical study. A detailled description of test inputs can be found on the project's Web site (Version numbers in parentheses indicate GIT commit hashes).

domains. Our experiments measure the dynamic SCoP coverage (*DynCov*) for each program instance.

3.8 Experimental Setting

In our experiments, we use the Low-Level Virtual Machine (LLVM) compiler framework [25]. Its common representation during all stages of compilation is the *LLVM Intermediate Representation* (LLVM-IR). LLVM-IR is a strongly typed static single-assignment (SSA)-based language that can represent a wide variety of high-level languages. High-level languages are supported via different language frontends, which provide LLVM-IR as output, e.g., CLANG [2] for C/C++, RUBINIUS [1] for Ruby and PYPY [8] for Python. LLVM-IR can be optimized independently of platform and language. LLVM performs polyhedral optimization via the plugin POLLY [18, 19]. POLLY retrieves information about SCoPs from compatible loop nests found in the LLVM-IR. POLLY's SCoP detection is able to derive a polyhedral description of the loop nest when the restrictions of class *Static* are obeyed.

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

Table 3. Preoptimization used for all LLVM-IR files throughout the empirical study.

On top we built 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.

To assist POLLY in identifying SCoPs in LLVM-IR code, various preprocessing steps must be performed, as shown in Table 3. These steps correspond to a relevant subset of the optimizations of LLVM's optimization level O3 (irrelevant steps such as *jump-threading* were removed). In our experiments, we use the same steps.

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 [23]. We have created one further optimization step, *polly-region-simplify*, as shown in Table 3, which establishes the SESE property for every region found in the CFG.

We have chosen to 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 [26] or GPROF [15], we have implemented the instrumentation based on the Performance Application Programming Interface (PAPI) [12]. It allowed us to retrieve timing information precisely at the entry and exit edges of a SCoP with high-precision timers. Out of the 4 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 5.2.

Our measurements have been implemented as a Makefile-driven [3] 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 pre processing of these IR-files and apply optimizations as shown in Table 3 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 three different classes (*Class*). Thus, each sample program had to be run with three different binaries, one for each class, as introduced in Section 3.7. With the instrumentation, we were able to calculate the necessary *DynCov* for each program run.

Static's SCoP detection has been implemented by instrumenting all SCoPs detected by POLLY, without further modification of the detection process. Instrumentation was inserted only at transition edges between SCoPs (SESE). For *Dynamic* and *Extended*, 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 in a bottom-up fashion and fixed all possible failures 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 2(b). If we were able to fix a problem with the assumed run-time information, we proceeded with POLLY's SCoP detection and expanded the detected region to the maximal possible size.

All experiments were conducted on an Intel i5 M520 processor (2 physical cores, 2.40 GHz) with 4GB RAM. In order to stabilize the values of run time measurements, we ensured (using the FIFO scheduling class in Linux) that the programs measured could not be preepmted during execution.

3.9 Deviations

We experienced measurement bias in a few cases, caused by the introduction of sample program instances for every class. Since every instance required different SCoPs to be instrumented, we experienced different function body alignment and cache effects, which led to minor inconsistent behavior of *DynCov* ($\textit{DynCov}_{\text{Stat}} > \textit{DynCov}_{\text{Dyn}}$). This circumstance does not bias our empirical results, as we discuss in Section 5.2.

4. Analysis

All results of our measurements are listed in Table 4 (at the end of the paper). The complete set of experimental results is available at the project's Web site.

4.1 Descriptive Statistics

Figure 3 shows the distributions of dynamic coverage of the three classes *Static*, *Dynamic*, and *Extended*; Table 5 lists the corresponding values of mean, variance, and standard deviation. At first glance, the dynamic coverage of *Static* is lowest, followed by *Dynamic* and *Extended*, but the variance of *Dynamic* and *Extended* is considerably higher than of *Static* – an observation that we discuss in Section 5.

Class	μ	s	s^2
Static	10	14	190
Dynamic	29	26	680
Extended	39	26	650

Table 5. Mean (μ), standard deviation (s), and variance (s^2) for all three classes.

Table 6 lists the mean dynamic coverages as well as relative differences between the three classes on a per-domain basis. An immediate observation is that the coverages and their relative differences differ considerably between individual domains. We will discuss this in Section 5.

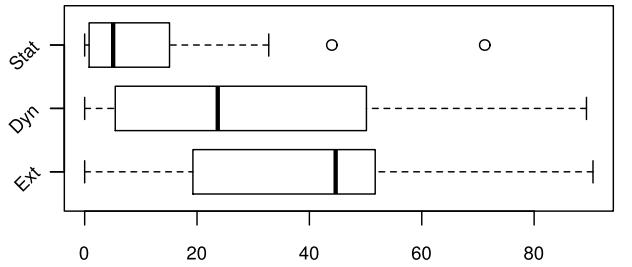

Figure 3. Distributions of *DynCov* for every class in form of boxplots. The left end of each box represents the lower quartile (25th percentile). The right end of each box represents the upper quartile (75th percentile). The left/right whiskers mark the lowest/highest datum within 1.5 interquartile range of the lower/upper quartile. The band inside each box denotes the median. Outliers are represented by circles.

Domain	μ_{Stat}	Δ	μ_{Dyn}	Δ	μ_{Ext}
Compilation	6.8	14	21	11	32
Compression	9.3	5.1	14	25	40
Database	9.6	3.2	13	16	29
Encryption	11	17	28	11	39
Multimedia	18	18	35	25	60
Scientific	4.2	37	41	0.23	42
Simulation	30	6.2	36	9.0	45
Verification	5.0	0.85	5.9	8.8	15

Table 6. Mean (μ) dynamic coverage and difference between domains (in %): Δ denotes the benefit between the corresponding left and right column.

4.2 Hypothesis Testing

Let us comment on the significance of our measurements. Our case study consists of three independent data sets (CLASS). A Shapiro-Wilk Test [33] on the dynamic SCoP coverage (DYNCOV) data of the classes *Static* ($p \ll 0.05$), *Dynamic* ($p < 0.05$) and *Extended* ($p \ll 0.05$) reveals that none of the three data sets is normally distributed.

With respect to $H_{1.1}$, a Mann-Whitney-U Test [27] reveals that $\textit{DynCov}_{\text{Dyn}}$ is significantly greater than $\textit{DynCov}_{\text{Stat}}$ ($p \ll 0.05$).

With respect to $H_{1.2}$, a Mann-Whitney-U Test reveals that $\textit{DynCov}_{\text{Ext}}$ is significantly greater than $\textit{DynCov}_{\text{Dyn}}$ ($p < 0.05$).

With respect to $H_{2.1}$, a Kruskal-Wallis Test [24] reveals that the benefit of *Dynamic* ($\textit{DynCov}_{\text{Dyn}} - \textit{DynCov}_{\text{Stat}}$) differs significantly accross different domains ($p \ll 0.05$). This is also shown in Table 6, e.g., the average benefit of Scientific (37 %) vs. the average benefit of Verification (0.85 %).

With respect to $H_{2.2}$, a Kruskal-Wallis Test reveals that the benefit of *Extended* ($\textit{DynCov}_{\text{Ext}} - \textit{DynCov}_{\text{Dyn}}$) differs significantly accross different domains ($p \ll 0.05$). Again, this can be seen in Table 6, e.g., the average benefit of Compression (25 %) vs. the average benefit of Scientific (0.23 %).

In summary, we can accept all of our hypotheses.

5. Discussion

Next, we describe the ramifications on polyhedral analysis and optimization of our results in detail.

5.1 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 3) of a program's total run time. Exceptions are SHA512, LULESH-OMP (Outliers in Figure 3) and POSTGRES: 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 ($\mathit{DynCov}_{\mathrm{Stat}} = 24\%$). While these outliers 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 polyhedral analysis (class *Dynamic*), *DynCov* rises to 5–50 % (as shown in Figure 3); this increase is significant ($H_{1.1}$). So, exploiting run-time information for polyhedral optimizations is of practical relevance.

Giving up on the restriction to affine linearity (class *Extended*) improves the overall benefit of polyhedral analysis even further to 19–51 %, as shown in Figure 3; this increase in *DynCov* is also significant ($H_{1.2}$). While this class serves only as upper bound for the potential of polyhedral optimization, which cannot reached with current models and implementations, it highlights the fact that there is still room for improving the model as well as implementations.

A notable observation is that the descriptive variance ($s^2 = 680$) of $\mathit{DynCov}_{\mathrm{Dyn}}$ is significantly greater ($p \ll 0.05$) than $\mathit{DynCov}_{\mathrm{Stat}}$ ($s^2 = 190$). According to hypothesis $H_{2.1}$, $\mathit{DynCov}_{\mathrm{Dyn}}$ differs significantly from $\mathit{DynCov}_{\mathrm{Stat}}$ across different domains. This suggests that the high variance is caused by a considerable difference between individual domains. Table 6 reveals that the domains Multimedia ($+18\,\%$) and Scientific ($+37\,\%$) gain the most when applying the polyhedron model at run time. This finding confirms the common belief that these domains are well-suited for polyhedral analysis. 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 analysis.

The same reasoning applies to the high variance of $\mathit{DynCov}_{\mathrm{Ext}}$. According to $H_{2.2}$, $\mathit{DynCov}_{\mathrm{Ext}}$ differs significantly from $\mathit{DynCov}_{\mathrm{Dyn}}$ across different domains. However, the greatest benefit could not only be achieved in the expected domains, as experienced in class *Dynamic*. Instead, giving up on the restriction to affine linearity, results in a significant increase in coverage in the domain Compression ($+25\%$). This shows that these show potential of being accessible with more sophisticated extensions to the polyhedron model in the future. In contrast, the domains that are expected to be well suited for polyhedral analysis could not always improve their *DynCov* to the same degree (e.g., Scientific $+0.23\,\%$). This suggests that the code regions that remain after detecting all SCoPs that belong to *Dynamic*, lack even statically known control-flow and are therefore out of reach of the polyhedron model anyway.

5.2 Threats to Validity

Construct validity Our measurement methodology results in different instrumentations per binary per class. This circumstance has an influence on the run-time fraction of SCoPs measured. The differing instrumentation is necessary because we have to detect the SCoPs based on the testing class and instrument each detected SCoP. They may not be a precise subset of each other in LLVM-IR. For each program of each class, we compared the difference between instrumented run time and the uninstrumented run time for each program to the expected overhead caused by our instrumentation.

As mentioned in Section 3.8, we measure virtual time with high-precision timers provided by the operating system. It is obvious that instrumentation causes higher run-time overhead than sampling. Mainly, this overhead is generated due to the instrumented calls to obtain timing information. However, it is possible that the instrumented code suffers from other negative side-effects as well, 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 is possible that it slows down a SCoP more than other program parts, which would increase *DynCov*. This did not cause problems during our experiments. We verified that by comparing the average run-time overhead (calibrated separately) of one instrumentation call and the actual average run time overhead of one instrumentation call (measured on each sample program).

Internal validity Furthermore, we depend on the quality of our input parameters, because we measure run time. We have taken care of 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 [**?**] for a JavaScript engine. This does not remove the dependence from *Input*, but we assume that the developer's own test cases catch the important code paths and the default benchmarks of a domain test the most common use cases, that is, we assume the benchmarks reflect the real world correctly.

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

5.3 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 will constantly flow into these tools and improve the situation (e.g., correct handling of integer wrapping and multi-dimensional arrays in LLVM).

Having said that, our study demonstrates the potential of applying polyhedral optimizations at run time. Using run-time information increased the dynamic coverage substantially. We consider this observation a major result and an important sign to the community to follow this path, both in terms of refining and extending the model and by extending practical tools accordingly.

Giving up on the restriction to affine linearity unfolds the full potential of polyhedral optimization that is theoretically possible and in reach of the polyhedron model; our results demonstrate that there is considerable room to extend the polyhedron model and its implementations.

6. Related Work

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

6.1 Alternative Studies

Alnaeli et. al. [4] conducted an empirical study on the parallelizability of open source systems. In their work they analyzed 11 open source software systems for their past and current parallelization opportunities. They conclude that the main problem with the programs tested were function calls inside the loop's 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.

6.2 Alternative Extensions

The following extensions focus on extending the polyhedron model's reach beyond affine linearity at compile time.

Benabderrahmane et al. [7] allows modeling arbitrary, non-recursive, control flow within a SCoP at compile time. This is done by converting control dependences to data dependences, if necessary. The same extension can be used to deal with while-loops in SCoPs. The while-loop is transformed into an unbounded for-loop and an exit-conditional is introduced in the body in form of a write access. 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 introduced dependence to the exit-conditional enforces a sequential schedule.

In contrast to stretching the modeling capabilities by giving up precision, there are a few extensions to the 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 the modeling, the transformation and the code generation at compile time by using real quantifier elimination [20]. Second, cylindrical algebraic decomposition [20] can be used to provide support for input programs that feature more complicated non-linearity, like polynomials in the index variables. However, both approaches suffer from significant performance penalties in the code synthesis as well as in the generated code itself, due to the more complex modeling and transformation phase.

6.3 Alternative Technologies

In addition to 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. Hence, a syntactic markup of SCoPs is required. LooPo [17] was the first such system. A recent system is PoCC [30], which implements a full compiler toolchain for automatic polyhedral optimization. It supports two polyhedral transformation tools: PLuTo [9, 10] and LeTSeE [28, 29]. The PLuTo scheduling algorithm focusses on providing a transformation that optimizes data locality on shared memory systems. In contrast to generating the optimal solution, LeTSeE uses an iterative approach by exploring the legal transformation space and tries to converge on the optimal solution.

Recently, implementations of the polyhedron model working on a compiler's intermediate representation (as does POLLY) have become available. The main advantage is that, unlike the source-code-based tools, SCoPs need not be written in a fixed syntactic form, since the loop structure and array accesses are obtained from loop and pointer analyses on the IR. A project similar to POLLY is the GCC project GRAPHITE [34]. Other implementations working on IR include the Wrap-IT [16] research project (based on Open64) and the IBM XL compiler [11].

Research on the field of run–time-based polyhedral optimization has started to emerge. The latest contribution by Jimborean [22] merges speculative techniques, like the LRPD test [31], and adaptive compilation with polyhedral optimization. However, most of the polyhedral optimization is still performed at compile time.

7. 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 dynamic coverages when applied to real-world programs at compile time (10 %). But, we found that 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). The time is ripe for researchers and tool builders to tap this potential. 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 sample systems and benchmarks are a good start for a community effort to coordinate efforts in improving polyhedral optimization.

References

[1] Rubinius: An environment for the Ruby programming language. `http://rubini.us`.

[2] Clang: A C language family frontend for LLVM. `http://clang.llvm.org`.

[3] GNU make. `http://www.gnu.org/software/make/`.

[4] S. M. Alnaeli, A. Alali, and J. I. Maletic. Empirically examining the parallelizability of open source software systems. In J. Cordy, T. Dean, D. Poshyvanyk, and R. Oliveto, editors, *WCRE*. IEEE Computer Society, 2012. to appear.

[5] 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 *IPDPS*, pages 1–12. IEEE Computer Society, 2008.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16. IEEE Computer Society, 2004.

[7] 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, *CC*, volume 6011 of *LNCS*, pages 283–303. Springer, 2010.

[8] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing jit. In S.-C. Khoo and J. G. Siek, editors, *PEPM*, pages 43–52. ACM, 2011.

[9] 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 L. J. Hendren, editor, *CC*, volume 4959 of *LNCS*, pages 132–146. Springer, 2008.

[10] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In Gupta and Amarasinghe [21], pages 101–113.

[11] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In V. Salapura, M. Gschwind, and J. Knoop, editors, *PACT*, pages 343–352. ACM, 2010.

[12] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[13] J.-F. Collard. Automatic parallelization of *while*-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[14] P. Feautrier and C. Lengauer. Polyhedron model. In D. Padua et al., editors, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

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

[16] 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*, 34:261–317, 2006.

[17] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In D. C. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *LCPC*, volume 1239 of *LNCS*, pages 603–604. Springer, 1997.

[18] T. Grosser, H. Zheng, R. Alor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly - polyhedral optimization in llvm. In C. Alias and C. Bastoul, editors, *IMPACT*, 2011.

[19] T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

[20] 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.

[21] R. Gupta and S. P. Amarasinghe, editors. *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008. ACM.

[22] 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.

[23] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *PLDI*, pages 171–185, 1994.

[24] W. Kruskal and W. Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, 47(260): 583–621, 1952.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–86. IEEE Computer Society, 2004.

[26] J. Levon. OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net/doc/`, 2007.

[27] 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*, 18(1):50–60, 1947.

[28] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *CGO*, pages 144–156. IEEE Computer Society, 2007.

[29] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In Gupta and Amarasinghe [21], pages 90–100.

[30] 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 *SC*, pages 1–11. IEEE Computer Society, 2010.

[31] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 10(2):160–180, 1999.

[32] L. Rauchwerger, N. M. Amato, and D. A. Padua. A Scalable Method for Run-Time Loop Parallelization. *Parallel Programming*, 23(6):537–576, 1995.

[33] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[34] 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 D. Nuzman and G. Fursin, editors, *GROW*, pages 1–13, 2010. `http://ctuning.org/workshop-grow10`.

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

Table 4. Dynamic SCoP coverage of the 3 different analysis classes: *Static*, *Dynamic*, *Extended* (in %). The $\mathbf{T}_{\text{Raw}}$ column 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 of this program in the respective class (in s).