On the Variety of Static Control Parts in Real-World Programs: from Affine via Multi-dimensional to Polynomial and Just-in-Time

Andreas Simbürger
Department of Informatics and Mathematics
University of Passau
andreas.simbuerger@uni-passau.de

Armin Größlinger
Department of Informatics and Mathematics
University of Passau
armin.groesslinger@uni-passau.de

ABSTRACT

The polyhedron model has been used successfully for automatic parallelization of code regions with loop nests satisfying certain restrictions, so-called static control parts. A popular implementation of this model is Polly (an extension of the LLVM compiler), which is able to identify static control parts in the intermediate representation of the compiler. We look at static control parts found in 50 real-world programs from different domains. We study whether these programs are amenable to polyhedral optimization by Polly at compile time or at run time. We report the number of static control parts with uniform or affine dependences found and study extensions of the current implementation in Polly. We consider extensions which handle multi-dimensional arrays with parametric sizes and arrays represented by "pointer-to-pointer" constructs. In addition, we extend the modeling capabilities of Polly to a model using semi-algebraic sets and real algebra instead of polyhedra and linear algebra. We do not only consider the number and size of the code regions found but measure the share of the run time the studied programs spend in the identified regions for each of the classes of static control parts under study.

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

Polyhedral Compilation, JIT, Loop Parallelisation, LLVM, Polly

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://impact.gforge.inria.fr/impact2014

1. INTRODUCTION

In recent years, multi- and many-core processors have become ubiquitous. The variety of parallel hardware architectures in different CPUs, GPUs, specialized accelerators, etc. has consequently given rise to a plethora of new programming languages, programming paradigms and parallel libraries. Although many of these approaches have been shown to be very successful, the success often comes at the price of requiring programmers to learn a new language, paradigm or library for each new hardware (such as GPUs) and each problem domain, e.g., linear algebra or stencil computations.

Among the new languages, paradigms and libraries there exist both high-level and low-level approaches. Both have their respective difficulties. As an example of a low-level approach consider CUDA or OpenCL for general-purpose programming on graphics cards. Here, the programming language has been quite stable over several hardware generations but programmers are required to rewrite their codes for newer hardware generations that have different requirements on the code to exhibit high performance. On the other hand, high-level approaches specific to a certain domain do not require modification of source codes across different hardware since the knowledge of how to achieve high performance is embedded in the code generator delivered with these systems; instead, users are required to learn a different system with its rules for syntax, semantics, etc. for each new domain.

Therefore, the success of these technologies does not make the aim of achieving automatic parallelization of general-purpose codes obsolete. Naturally, this is a difficult goal but advances towards making automatic parallelization successful for real-world codes, i.e., in production environments where the parallelizing compiler cannot make optimistic assumptions about the code being transformed (such as the absence of corner cases), have been made in recent years. The projects Graphite [27] and Polly [16] have brought parallelization based on the polyhedron model [10] to GCC [11] and LLVM [21], respectively. Both systems work on the intermediate representation level, i.e., they are independent of the source language. This brings the advantage of providing polyhedral optimizations to all the languages supported by the compiler framework but, on the other hand, has the disadvantage that, in general, less information to help the parallelization process is available.

Over time, the polyhedron model has been extended, originally limited to uniform dependences, then affine dependences and recently even beyond the restriction to polyhedra as mathematical device. On the other side, working on the

intermediate representation instead of source code brings new challenges for compilers based on polyhedral analysis. We analyze real-world programs from different domains to find out how much of the programs' run times are spent in code regions which are amenable to different variants of polyhedral optimization (so-called static control parts).

This paper is organized as follows. We state our contribution in Section 1.1 and discuss related work in Section 1.2. In Section 2, we briefly introduce the polyhedron model and an extension based on polynomial expressions. Section 3 defines several classes of code regions that fit different variants of the model. Sections 4 presents the programs used in our study and we report and discuss the results of our analysis before we conclude in Section 5.

1.1 Contribution

We are interested in the question of how much potential there is for automatic algebraic compilation frameworks in real-world programs. As a measure for the potential we do not use the number of lines in the source code or the number of loops which can be considered for automatic parallelization; instead, we want to determine the actual share of the run time which a given program spends in code regions that are amenable to optimization. A central unit of optimization in the field of polyhedral compilation is the static control part (SCoP, cf. Section 2). The number of SCoPs that can be optimized depends on the choice of the mathematical framework (usually polyhedral, but we consider extensions thereof) and the time of the optimization (compile time or run time).

In our own previous work [26], we have studied how much of the run time of real-world programs is amenable to polyhedral optimization (using Polly) at compile time compared to how much is amenable to polyhedral optimization at run time, e.g., using a just-in-time compiler. With the present work, we contribute our analysis concerning

- how many of the SCoPs detected at compile time have uniform dependences compared to affine dependences,
- whether it is possible to detect multi-dimensional arrays represented by pointer-to-pointer constructs (instead of using a contiguous memory block for the multi-dimensional array),
- an algorithm to detect some multi-dimensional array accesses (not represented by pointer-to-pointer constructs) from the linearized array representation used in LLVM, and how many such multi-dimensional array accesses can be reconstructed in the programs studied,
- the run-time share amenable to analysis based on semi-algebraic sets (instead of polyhedral analysis).

In addition, we repeat some of the experiments from our previous work [26] with the current versions of LLVM, Clang (LLVM's C language frontend) and Polly[1] for comparison with the other numbers in the present work.

1.2 Related Work

Not so many studies about the potential for automatic parallelization over a variety of real-world programs have been conducted so far. One such study has been done by Alnaeli et al. [1]. They studied the parallelizability of 11 open-source software systems and concluded that function calls with side-effects inside loops are a major problem but they did not analyze the fraction of the run time spent in loops.

Doerfert et al. [9] already experimented with the application of the polyhedron model at run time. In their work, they implemented a subset of our class *Dynamic*, i.e., dealing with non-affine parameters, and showed interesting speedups on the Polybench benchmark suite.

Several approaches are viable to increase the run-time fraction amenable to automatic optimization. One is to stretch the modeling capabilities of the polyhedron model. Griebl [13] presented an extension which models **while** loops as unbounded **for** loops and additional write accesses. Benabderrahmane et al. [2] model arbitrary, non-recursive, control flow within a SCoP at compile at the cost of introducing data dependences where control dependences may exist. The data dependences introduced in these approaches reduce the potential for parallelization. In contrast, we study the applicability of the polyhedron model at compile time and at run time, and a more powerful compile time model based on semi-algebraic sets.

There are many well-known polyhedral optimizers/parallelizers which work on source code, e.g., [14, 4, 25] and several others. But these tools need additional information (apart from the source code) because the correctness of the parallelization depends on non-trivial properties of the code, e.g., the absence of pointer aliasing or the side-effect freeness of functions called. Usually, these properties have to be asserted by the user or the tool optimistically relies on their validity. In contrast, we aim at characterizing the potential of the approaches we compare on real-world codes; therefore, we work on intermediate code and non-trivial properties (such as the absence of pointer aliasing) must be proved by the compiler's program analyses before relying on them. There are other compiler frameworks apart from LLVM which have interfaces to polyhedral analysis and optimization on the intermediate representation level, e.g., GCC via Graphite [27], Open64 via Wrap-IT [12], R-Stream [20] and the IBM XL compiler [3].

Run-time polyhedral optimization is becoming a topic of recent studies, such as by Jimborean [19] which merges polyhedral and speculative optimization techniques. In our own previous work, we conducted a study focusing on the potential of just-in-time polyhedral optimization [26].

Recognizing multi-dimensional array accesses in the intermediate representation (where accesses are linearized) is, in general, a hard problem. There has been some discussion [15] and, very recently, first patches have been proposed [24] for Polly to start dealing with this problem. This is independent of our presented work as we use real algebra to estimate how many such accesses can be discovered by the heuristic we propose. Maslov [22] proposed an algorithm to "delinearize" affine dependence equations by breaking them into several equations. This algorithm works on the coefficients of the loop iterators occurring in the dependence equations and uses integer operations such as greatest common divisor on these coefficients; therefore, it is not obvious whether the presented technique can be applied to recognize multi-dimensional array accesses with parametric array sizes. Maslov and Pugh [23] also presented an algorithm to simplify certain polynomial dependence constraints to affine constraints. This

[1]Current on 2013-10-24.

technique and symbolic Bernstein expansion [7] may provide alternatives to our use of quantifier elimination in the real numbers to test our sufficient condition for delinearization (cf. Section 3.5).

Wu et al. [29] presented an algorithm to perform instance-wise alias analysis for arrays represented by pointer-to-pointer constructs. Our results (cf. Section 4.3) show that such techniques are needed to increase the applicability of polyhedral compilation on real-world codes.

2. BACKGROUND

2.1 Polyhedron Model

The polyhedron model [10] tries to capture the essence of a loop nest with n loops and statements inside the nest by representing the iterations of the loops and the data dependences between loop iterations as sets and relations over an n-dimensional space. Each loop iteration corresponds to a point $\vec{i} \in \mathbb{Z}^n$ where the components of $\vec{i}$ correspond to the values of the loop iterators. The set of all iterations for which a statement executes is called the domain of the statement. For illustration, such a loop nest is shown in Figure 1. Here, the domains for the two statements S and T are given by $D_T = \{(i,j) : 1 \leq i \leq n \wedge i \leq j \leq n\}$ and $D_S = D_T \cap \{(i,j) : i \geq n - j\}$.

By transforming the domains, i.e., changing the shape of the sets and the enumeration order of the points, optimizations can be expressed. For example, parallelization means that, after transformation, one or several dimensions are mapped to a parallel loop instead of a sequential one. To guarantee the correctness of the transformed code, the transformation must not violate the data dependences of the original code, i.e., the order of operations accessing the same memory cell (involving at least one write access) must not be changed. This order is expressed as a relation on the iteration domains which relates dependent operations (the later access in the original code) to its source. The power of the polyhedral model comes from the fact that one can perform an optimizing search for the best transformation maximizing a given objective function, e.g., to maximize parallelism, under the constraint that dependences are not violated.

Code regions which can be described in terms of domains and dependence relations are called static control parts (SCoPs). "Static control" refers to the fact that the control flow and data dependences can be computed at compile time and described by a finite set of (parametric) expressions. To compute the domains and dependences, the mathematical formalism used must be decidable, i.e., it must be possible to algorithmically compute the dependences, solve the optimization problem which describes the transformation one wants to perform, and generate code from the transformed model.

The computability requirement is a hard constraint. The polyhedron model derives its name from the fact that when one uses affine expressions (linear expressions plus a constant) in the loop bounds and one only allows arrays with affine subscripts, then linear algebra and integer linear programming can be used to compute dependences, find optimizing transformations and generate code for the transformed model. The restriction to affine expressions gives the model its name, polyhedron model, as the domains and dependence relations are, geometrically, ($\mathbb{Z}$-)polyhedra.

In the example in Figure 1, determining the dependences by pure polyhedral means is not possible. To determine the

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

Figure 1: A (non-affine) static control part (SCoP)

data dependences between statements S and T induced by array B, one has to solve the equation $n \cdot i = i'$ (considering iteration (i,j) of S and (i',j') of T) w.r.t. the constraints given by the domains of S and T. Due to the parametrization with n in the term $n \cdot i$, the equation cannot be solved by affine methods and, hence, the problem is not polyhedral. To come around this complication, different approaches are possible:

- Polyhedral analysis can be performed at run time (instead of compile time). Then, the value for the parameter n is known and the problem becomes affine.
- If the non-affine array subscript expression is equivalent to a multi-dimensional subscript, then polyhedral analysis can be performed (given that an equivalent multi-dimensional expression can be computed). In the example, the two accesses `B[n*i]` and `B[i']` can equivalently be modeled by two-dimensional accesses since i' is in the range $[1,n]$, i.e., the minimal and maximal values for i' differ by less than n. Therefore, the accesses are equivalent to a two-dimensional array with an inner dimension with n elements. But since the range does not start at 0, the actual modeling is not immediately obvious (e.g., it is not `B[i][0]` and `B[0][i']` as this implies $i = i' = 0$ but that is not the solution of the original problem which is $i = 1$, $i' = n$). Since constructing equivalent multi-dimensional arrays accesses is non-trivial and, to be able to compute dependences, not strictly necessary, we restrict our attention to developing a sufficient condition which allows us to delinearize the dependence equation (cf. Section 3.5).
- A more powerful mathematical framework is used. A formalism able to deal with polynomials instead of affine expressions and satisfying the computability criterion can be used to handle such non-polyhedral problems.

We study all three approaches (with quantifier elimination in the real numbers as a non-polyhedral formalism), cf. Section 3. We now briefly introduce semi-algebraic sets and quantifier elimination.

2.2 Semi-Algebraic Sets and Quantifier Elimination

The restriction to affine expressions guarantees that modeling, transformation and code generation are computable. Instead of affine expressions, one can allow arbitrary polynomials in the iterators and problem parameters, i.e., loop bounds and array accesses such as $n \cdot i$ for an iterator i and a parameter n are then admissible. The drawback of this approach is that, in general, it is not possible to compute the integral solutions of polynomial equality and inequality systems because this would imply the solvability of Hilbert's

10^{th} problem (which is known to be unsolvable [8]). In previous work [17], we have shown that it is nevertheless possible to model loops, transform them and generate target code for the transformed loops.

The main tool in this generalization of the polyhedron model is quantifier elimination in the real numbers. Quantifier elimination means that for any formula φ with universal and existential quantifiers and the usual logical connectives ($\wedge$, $\vee$, $\rightarrow$, $\neg$) over polynomial (in)equalities, there exists an equivalent formula without quantifiers. For example, to determine whether dependences exist between the array accesses to B in Figure 1 (with the access in S happening before the access in T, i.e., $i \leq i' - 1$), we can use the following formula:

$$\exists i \exists j \exists i' \exists j' \left(n \cdot i = i' \wedge (i, j) \in D_S \wedge (i', j') \in D_T \wedge i \leq i' - 1 \right).$$

Applying quantifier elimination will give us an equivalent formula without quantifiers, i.e., the only remaining indeterminate is n, so the result gives the conditions on n for the existence of dependences (here: $n \geq 2$). By using quantifier-elimination with answer (i.e., solutions for the existentially quantified variables are computed) we can also get a description of the dependences (here: $i = 1$, $i' = n$); for details we refer to the literature [17].

Quantifier elimination can be applied to arbitrary formulas (as stated above) but is an expensive operation (doubly exponential in the number of quantifiers in the worst case). We use QEPCAD [5] to perform quantifier elimination. When an elimination problem happens to be affine, we call Verdoolaege's integer set library [28] to get the result faster (and profit from its ability to compute in the integers instead of the real numbers). In our experimental study, quantifier elimination needed up to several seconds to solve the problems given to it arising from our algorithm in Section 3.5.

2.3 Polly

We have chosen to use POLLY to analyze the programs we study as we are aiming for real-world programs and do not want to make any optimistic assumptions about the code a compiler does not have access to. Since POLLY is based on LLVM, it inherits some limitations from LLVM that influence how many SCoPs can be detected:

- POLLY uses LLVM's scalar evolution analysis to compute loop bounds and array subscripts. LLVM's implementation of scalar evolution does not seem to handle all cases of polynomial expressions.

- Array subscripts are linearized in the intermediate representation, i.e., there are, a priori, no multi-dimensional arrays with parametric sizes for the inner dimensions in POLLY. When the sizes of inner dimensions are fixed numbers, a multi-dimensional array access can trivially be represented by a single affine subscript (i.e., $A[i][j]$ is equivalent to $A[100 \cdot i + j]$ when the inner dimension of A is declared to have 100 elements) and POLLY can model these cases.

- POLLY relies on LLVM's alias analysis to check whether two memory accesses (derived from different base pointers) can alias or not. The effectiveness of alias analysis also depends on whether the language frontend passes the information it has over to the intermediate representation.

When presenting our classification of SCoPs (Section 3) we refer to the respective limitation that is relevant for the given class.

For multi-dimensional array accesses the question is how to recognize them. In some cases, the sequence of instructions in the intermediate representation reveals the multi-dimensional nature of an access. For example, the C language frontend CLANG uses the following sequence to implement the address computation for an array access `A[i][j]` (where `A` is declared as `float A[][n]`):

```
1   %0 = mul nsw i32 %i, %n
2   %idx = getelementptr float* %A, i32 %0
3   %idx1 = getelementptr float* %idx, i32 %j
```

The two address computations for the row address `A[i]` (i.e., `n*i+A` in `%idx`) and for `A[i][j]` (in `%idx1`) make it quite "obvious" that the access is two-dimensional. But turning on optimizations (e.g., using the `-O1` option of CLANG) changes the sequence into:

```
1   %0 = mul nsw i32 %i, %n
2   %idx.s = add i32 %0, %j
3   %idx1 = getelementptr float* %A, i32 %idx.s
```

Here, the expression $n \cdot i + j$ is computed first and its value is used as an index into the array `A`. Therefore, it is not obvious anymore that this represents a multi-dimensional array access. This second method to compute the access to the array element is also found in code where multi-dimensional accesses have been linearized by hand. Following this observation, we do not try to recognize multi-dimensional array accesses in the intermediate representation directly but keep POLLY's use of scalar evolution analysis to find the address which is accessed. In both cases for the above example, LLVM's scalar evolution analysis computes the address as `n*i+j+A`. Working with these expressions has the additional benefit that overflows in the inner dimensions of the access (e.g., `A[0][n]` aliases `A[1][0]`) are captured by them ($n \cdot 0 + n$ equals $1 \cdot n + 0$) and we do not need to rely on (or try to check) that no overflows occur.

3. CLASSIFICATION OF STATIC CONTROL PARTS

The successful detection of SCoPs 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., to deal with non-affine expressions. We introduce five classes of SCoPs which we address and compare in our study. When we say that a class A is smaller than a class B, we mean that the code regions covered by B encompass the regions covered by A. Formally, a class B encompasses class A if every SCoP $a \in A$ is either an element of B or there exists a SCoP $b \in B$ such that b encompasses the code region of SCoP a. This implies that when class B encompasses class A the number of SCoPs in B can be less than the number of SCoPs in A (because several SCoPs of class A are covered by one SCoP in class B). In order to be able to compare the sizes of the different classes in our study, our implementation does not combine the SCoPs that belong to class *Static* with other SCoPs when the SCoPs of a bigger class are determined. Therefore, the SCoPs in all classes apart from *Static* are not always maximal (as is usual in the definition of SCoPs). This is irrelevant for our study.

When applying optimizations to SCoPs one would naturally prefer SCoPs to be as big as possible.

3.1 Class Static

This class covers all SCoPs that are found by Polly (without modifications) on the intermediate representation. All arrays detected in SCoPs are one-dimensional, i.e., multi-dimensional arrays are only found when the sizes of the inner dimensions are compile-time constants. Arrays represented by pointer-to-pointer constructs are not considered. In our analysis, we also determine the SCoPs within this class which have only uniform dependences (i.e., the distance between source and target of a dependence is constant). SCoPs with uniform dependences have stronger properties and require less expensive algorithms to transform.

3.2 Pointer-to-Pointer Arrays (Class Ptr)

This class is an extension of class *Static* which allows pointer-to-pointer constructs for multi-dimensional arrays. On the implementation side, this only requires a few changes in Polly. The main restriction for valid modeling is the absence of aliasing between the pointers in outer array dimensions. Consider a pointer to pointer `A` and a statement using this pointer:

```
1  float **A;
2  A[i][j] = A[i-1][j-1];
```

`A` can only be modeled as a two-dimensional array when it is known that the inner arrays do not alias, i.e., the arrays pointed to by `A[i]` for different i do not overlap each other. In addition, the outer dimension of `A` should not alias with other pointers/arrays. In the C language, this can be expressed by using the `restrict` keyword,[2] i.e., by declaring the pointer as

```
1  float *restrict *restrict A;
```

Unfortunately, the current version of LLVM's C frontend (Clang) does not seem to take advantage of the `restrict` keyword except for function arguments and, for function arguments, only for the outermost dimension. Therefore, LLVM's alias analysis does not currently profit from the presence of `restrict` annotations in pointer-to-pointer constructs. This makes it unlikely to find actual instances of pointer-to-pointer arrays that can be modeled. For our study, we also determined an optimistic number for the pointer to pointer arrays by disabling Polly's check for aliasing.

3.3 Class Dynamic

To increase the number of SCoPs that can be handled, one can think of optimization at run time. Some code regions which cannot be proved at compile time to be SCoPs may turn out to be SCoPs at run time. This is due to the fact that at run time more information is available. In particular, three pieces of information can be exploited at run time:

1. Values of parameters. This allows to handle, e.g., expressions of the form $n \cdot i$, which are not affine at compile time but become affine as soon as the value for n is known.
2. Aliasing. The values for pointers are known at run time, i.e., it can be determined whether two arrays alias or not. In case of aliasing, both arrays can be modeled as a single array (with different, constant offsets into the single array).
3. Control flow and side effects. Control flow may depend on run-time values (other than parameters), and functions called inside a SCoP candidate may have side effects. In some cases, the control flow becomes static (in the sense of a SCoP) and the side effects of functions called can be modeled by polyhedral means when modeling is done at run time.

[2]The precise semantics of `restrict` is more complex but the details are not relevant here.

For a more elaborate discussion of class *Dynamic*, we refer the reader to our previous work [26]. In class *Dynamic*, we allow all code regions for which we can determine at compile time that they become SCoPs when reached at run time. Clearly, class *Dynamic* is bigger than class *Static*.

3.4 Class Algebraic

In this class, we permit all code regions that satisfy the same requirements as needed for class *Static*, with the exception that arbitrary polynomials in the loop iterators and parameters are allowed. This enables, for example, $n \cdot i$ in array subscripts and loop bounds. The name of this class comes from the fact that subsets of $\mathbb{R}^n$ defined by polynomial (in)equalities are called semi-algebraic.

To extract loop bounds and array subscripts we rely on the scalar evolution analysis present in LLVM (as does Polly). Due to limitations in the implementation, LLVM cannot currently detect products between iterators in the general case, i.e., i^2 is not supported. We consider this a minor limitation for class *Algebraic* as we assume that products between iterators rarely occur in practice. With this limitation, class *Algebraic* is expected to lie between classes *Static* and *Dynamic*.

3.5 Multi-dimensional Array Accesses

A major improvement of Polly would be to support multi-dimensional arrays (with contiguous layout, i.e., not represented by pointer to pointer constructs). Multi-dimensional array accesses are linearized in the intermediate representation, e.g., `A[i][j]` is represented as `A[n*i+j]` with `n` being the size of the inner dimension. Reconstructing multi-dimensional accesses from their linearization is non-trivial. For example, `A[n*i+i+j]` could be `A[i][i+j]` with the inner dimension having a size of `n` or `A[i+1][j]` with the inner dimension having size `n+1` (because `n*i+i+j` equals `(n+1)*i+j`). The delinearization performed must be consistent across all the accesses concerned, i.e., the sizes of the inner dimensions must be the same, and must take care of overflows in the inner dimensions' indices.

We do not give a solution for the general delinearization problem. Instead, we propose sufficient conditions for reducing the dependence analysis problem to affine conditions. For dependence analysis, it is sufficient to deal with two array accesses at a time. Consider two accesses to the same array with linearized subscripts a and a', both polynomials (with integer coefficients) in the iterators $\vec{i}$ and $\vec{i'}$, respectively, and the parameters. To compute the dependences, one has to solve the equation $a = a'$ (under the additional constraint that the iterators lie in their respective domains, i.e., $\vec{i} \in D$

and $\vec{i'} \in D'$). Our proposed heuristic tries to rewrite $a - a'$ as a sum of products

$$a - a' = \sum_{x=1}^{k} \pi_x \gamma_x \quad (1)$$

where the π_x are polynomials in the parameters and the γ_x are affine expressions in the iterators, both with integer coefficients, and the summands $\pi_x \gamma_x$ obey the following total ordering condition:

$$\forall \vec{i} \in D, \vec{i'} \in D' : |\pi_x \gamma_x| \leq |\pi_{x+1}| - 1 \text{ for } 1 \leq x < k \quad (2)$$

When (1) and (2) hold, $a - a' = 0$ is equivalent to

$$\gamma_1 = 0 \wedge \cdots \wedge \gamma_k = 0$$

because the π_x and γ_x are integer-valued expressions. Since the γ_x are affine expressions, this system can be solved using polyhedral methods.

The obvious prerequisite is that $a - a'$ does not contain products between iterators (which are unlikely). To derive the π_x and γ_x, we apply a three step heuristic:

1. Split the polynomial $a - a'$ into its terms, i.e., $a - a' = \sum_{x=1}^{l} t_x$ where each t_x is a product of iterators and parameters (and a constant).
2. Group terms by their parameters (as much as possible), i.e., $a - a' = \sum_{x=1}^{m} \rho_x \gamma_x$ where each ρ_x is a product of parameters (or a constant) and each γ_x is an affine expression in the iterators. The GCD of the coefficients in γ_x should be 1 (i.e., a common constant factor of the coefficients is moved to ρ_x).
3. Factor out common γ_x (as much as possible), i.e., $a - a' = \sum_{x=1}^{k} \pi_x \gamma_x$ where π_x are polynomials in the parameters.

After deriving the π_x and γ_x, the total ordering required by (2) has to be checked. Since the expressions $\pi_x \cdot \gamma_x$ are not affine (when $a - a'$ is not affine) we cannot use integer linear programming to check these conditions in general. But it is possible to use quantifier elimination in the real numbers to do so (Bernstein expansion [7] or the technique of [23] may be used alternatively but we have not tried these approaches). We give the quantifier elimination the decision problem

$$\forall \vec{i}, \vec{i'}, \vec{p} (\vec{i} \in D \wedge \vec{i'} \in D' \rightarrow |\pi_x \gamma_x| \leq |\pi_y| - 1)$$

to check if $\pi_x \gamma_x$ precedes $\pi_y \gamma_y$ in the final order. By asking several such questions we check that a total ordering of the $\pi_x \gamma_x$ as required by (2) exists. Using quantifier elimination in the real numbers ignores the integrality of the variables but as we check inequalities this is a minor loss of precision. Our heuristic works well at least for simpler cases, e.g., it correctly delinearizes the dependence equations for matrix-matrix multiply with parametric array sizes.

Example: Consider the two accesses `A[(n+2+m)*i]` and `A[i'+2*m+2*n]`. Here, $a - a'$ is $ni + 2i + mi - i' - 2m - 2n$ and the terms in Step 1 are ni, $2i$, mi, $-i'$, $-2m$, $-2n$. Step 2 (factoring out parameters) yields $a - a' = n(i-2) + m(i-2) + 1(2i - i')$. Step 3 (factoring out common expressions in the iterators, here: $i-2$) yields $a - a' = (n+m)(i-2) + 1(2i - i')$. Finally, we check if the loop bounds imply that $|1(2i - i')| \leq |n+m| - 1$ (formally, we would also check the (unlikely) case that $|(n+m)(i-2)| \leq |1| - 1$ holds). If so, then $a - a' = 0$ is equivalent to $i - 2 = 0 \wedge 2i - i' = 0$, i.e., $i = 2$ and $i' = 4$.

In our implementation, we find SCoPs belonging to class *Multi* by checking for SCoPs in class *Algebraic* if the iteration domains are affine and our heuristic finds that the differences between array subscripts required for dependence analysis satisfy the conditions of (1) and (2). Therefore, class *Multi* lies between classes *Static* and *Algebraic*.

4. STUDY

4.1 Measurement Methodology

We are interested in how much run time the programs we study spend in SCoPs of the different classes we have introduced. To estimate the potential for optimization in each class, we measure the fraction of the run time spent in SCoPs. We call this number (given in %) the *execution SCoP coverage* (*ExecCov*) (see also [26]).

Our experiments measure the execution SCoP coverage of each program in the classes *Static* ($ExecCov_{\text{Stat}}$), *Algebraic* ($ExecCov_{\text{Alg}}$), and *Dynamic* ($ExecCov_{\text{Dyn}}$) using instrumentation. We have chosen instrumentation based on the Performance Application Programming Interface (PAPI) [6]. It allows us to retrieve timing information precisely at each entry and each exit edge of a SCoP with high-precision timers. Among the different clock variants offered by PAPI, we have chosen virtual time for our measurements. Virtual time consists of a process's actual execution time in user mode (user time) and time spent in privileged mode (system time). Therefore, using virtual timers we measure the time a process actually spends executing, both for the total run time and for the time spent in SCoPs. We have calibrated the overhead of our instrumentation and subtracted the overhead before computing the ratio between the time spent in SCoPs and the absolute run time of a program. Due to fluctuations in run times between different runs and the slight influence of the instrumentation on the run times (after subtracting the instrumentation overhead) it is possible that, for a particular program, an *ExecCov* number for a bigger class is reported to be slightly lower than that of a smaller class.

In addition to the execution SCoP coverage, we also report the number of SCoPs detected in each class and, for class *Static*, the number of SCoPs with no dependences at all, with uniform dependences only and with arbitrary affine dependences.

4.2 Programs Studied

The programs under study were selected from an unbiased selection of open-source programs. We considered programs of seven domains, of which not all are typical targets of polyhedral optimization (Compilation, Compression, Database, Verification).

The programs have been run with the test inputs supplied with the program package. In case of more than one input for a particular program, the measurements have been aggregated over the different runs. The absolute run times of the programs range from 105 microseconds to 700 seconds. Due to the use of high-precision timers with nanosecond resolution we can assume that, even for the shorter program runs, measurements are meaningful. All measurements were performed on an AMD Phenom II X4 965 CPU and 8GB RAM under Linux with the measured process running in schedule class `SCHED_FIFO` to minimize the influence of other processes on the system.

4.3 Results

The aggregated results can be found in Table 1. The abbreviated column names show the number of SCoPs and dependences as well as the execution SCoP coverages found in our experiments.

The numbers for classes *Static* and *Dynamic* differ from the numbers given in our previous study [26] and not all programs are present. This is due partly to changes in POLLY and LLVM that affect the number of SCoPs detected and whether we can run the programs with our infrastructure correctly. The detection of SCoPs is very sensitive to the selection and ordering of preparational phases run before the actual SCoP detection. For example, the current version of POLLY does not rely on SCoPs to be in single-entry single-exit regions anymore but the lifting of this restriction can make the construction of canonical induction variables more difficult, leading to changes in the number of SCoPs detected.

The first group of columns shows the number of SCoPs found in the different classes:

S_{st}**:** No. of SCoPs found in class *Static*.
S_{pp}**:** No. of SCoPs with pointer-to-pointer array accesses (i.e., the increment of *Ptr* over *Static*).
S_{ppa}**:** As S_{pp}, but without aliasing checks.
S_{md}**:** No. of SCoPs containing multi-dimensional arrays (i.e., the increment of class *Multi* over *Static*).
S_{na}**:** No. of non-affine SCoPs (i.e., the increment of class *Algebraic* over *Static*).
S_{jit}**:** No. of SCoPs belonging to class *Dynamic* but not to class *Static*.

Note that S_{pp}, S_{ppa}, S_{md}, S_{na} and S_{jit} give the number of *additional* SCoPs compared to *Static*.

The second group of columns shows detailed information about the dependences found in class *Static*:

S_{nd}**:** No. of SCoPs without dependences in class *Static*.
S_{u}**:** No. of SCoPs with uniform dependences only in class *Static*.
S_{a}**:** No. of SCoPs with affine dependences in class *Static*.
d_{u}**:** No. of uniform dependences in class *Static*.
d_{a}**:** No. of (non-uniform) affine dependences in class *Static*.

Furthermore, the third and fourth group of columns shows the *ExecCov* of uniform (t_{u}) and affine (t_{a}) SCoPs in class *Static*, as well as the *ExecCov* of all experiments in the three classes *Static*, *Algebraic*, and *Dynamic*:

t_{u}**:** *ExecCov* of uniform SCoPs.
t_{a}**:** *ExecCov* of (non-uniform) affine SCoPs.
Stat**:** *ExecCov* of class *Static*.
Alg**:** *ExecCov* of class *Algebraic*.
Dyn**:** *ExecCov* of class *Dynamic*.

When comparing the numbers of SCoPs found in the code against the values for *ExecCov*, one has to keep in mind that it is not guaranteed in our experiments that all SCoPs detected are executed at run time. Which SCoPs are executed depends on the program input and we decided to use the testinput(s) provided by the developers of the respective projects.

Figure 2 shows that the number of additional SCoPs detected in class *Algebraic* (compared to *Static*) is very low. This is against our expectation that expressions of the form `A[n*i]` are used at least occasionally in real-world code. Only in 10 out of 50 experiments a small number of non-affine expressions (between 2 and 37) was detected. Interestingly, the few SCoPs in class *Algebraic* for POVRAY cover 41% of the run time. Only the LAPACK programs (XEIG∗ and XLIN∗ in Table 1) and two codes in class Encryption exhibit a number of algebraic SCoPs comparable to the number of SCoPs in class *Static* but, unfortunately, these SCoPs do not seem to contribute to the execution coverage significantly. The subset *Multi* (which contains SCoPs with multi-dimensional array accesses according to our definition in Section 3.5) of *Algebraic* is even smaller. Only in 2 out of 50 experiments could we find *Multi* to be bigger than *Static* (by 1 and 3 SCoPs, respectively). Considering this, the increase of *ExecCov* in class *Dynamic* is possible because of sufficient information about the existence of aliasing and side-effects of function calls.

In contrast to this, 22 experiments out of 50 include more SCoPs, if support for multi-dimensional arrays based on pointer-to-pointer accesses is added (Class *Ptr*). The number of SCoPs detected is (optimistically) increased further when aliasing is ignored by POLLY.

The reasons for not detecting more polynomial expressions and multi-dimensional array accesses are not completely clear at the moment. We have done a preliminary investigation which suggests that the construction of canonical induction variables in POLLY is very sensitive to the optimization flags and preparational phases run before POLLY's SCoP detection. Making induction variable construction and the whole process of SCoP detection more robust is an aim for our further research. In addition, some codes (e.g., some LAPACK routines) take separate parameters for the size of the iteration domain and the array sizes. Here, we cannot statically prove that the accesses do not overflow and, hence, cannot consider the array accesses to be multi-dimensional. It would be possible to compute the conditions on the parameters required to make the accesses multi-dimensional using quantifier elimination but we have not studied this. Finally, improving alias analysis in LLVM to enhance the support for pointer-to-pointer constructs, e.g., by implementing the technique presented in [29], is also a topic for further study.

For two programs, we could not determine run times: LEVELDB uses threads extensively and this is currently not supported by our framework; LAMMPS could not be run due to technical complications when linking the final executable. BLOWFISH did not contain any SCoPs we could detect, so *ExecCov* is zero for all classes. The *ExecCov* numbers determined show that in several cases the share of the run time amenable to polyhedral optimizations can be increased significantly when SCoPs are detected at run time instead of compile time.

5. CONCLUSIONS

We have studied 50 programs from different domains and reported on how many static control parts (SCoPs) can be found with different techniques and how much of their run times the programs spend in SCoPs. We have based our work on LLVM and POLLY. We confirm our previous finding that performing polyhedral analysis at run time instead of at compile time can significantly increase the potential for polyhedral optimizations for some programs. For the compile time approach we have counted the number of SCoPs with no dependences, only uniform dependences and arbitrary affine dependences. As POLLY is currently limited to one-dimensional array accesses (with an affine subscript),

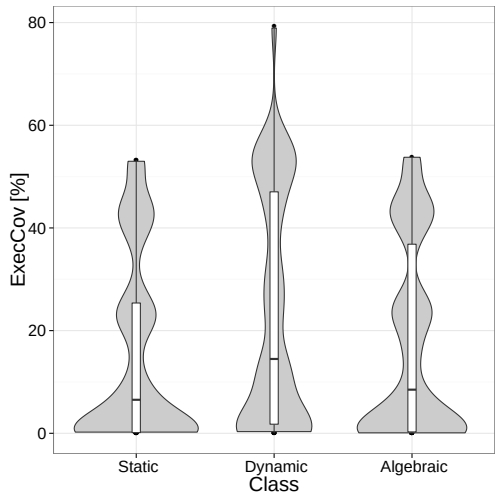

Figure 2: Distributions of *ExecCov* for *Static*, *Dynamic*, and *Algebraic* as violin plots [18] (combination of box plots and kernel density plots). The low end of the box represents the lower quartile (25 percentile), the top end the upper quartile (75 percentile). The bottom/top whiskers mark the 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*, *Dynamic*, and *Algebraic* cover the same area.

we explored whether one can target a bigger share of the run time by allowing three extensions: multi-dimensional arrays represented by pointer-to-pointer constructs, multi-dimensional arrays with linearized subscripts and SCoPs with arbitrary polynomials (instead of affine expressions) for the loop bounds and array subscripts. The numbers we observed in our measurements suggest that further research is necessary. We found that allowing pointer-to-pointer constructs may significantly increase the applicability of polyhedral optimization. Unfortunately, LLVM is currently not able to prove the absence of aliasing between the pointers to inner array dimensions; therefore, a significant number of pointer-to-pointer constructs is only detected when aliasing checks are turned off. We find that allowing polynomials instead of affine expressions does currenetly not give a significant increase (with a few exceptions) in the potential for optimization; this is surprising as one could expect linearized multi-dimensional arrays with parametric sizes to occur in practice. Our preliminary analysis of this situation suggests that the success of Polly's SCoP detection, especially for SCoPs with polynomial expressions, is very sensitive to the selection and ordering of preparational passes run. As a next step, we plan to research the detailed cases for not detecting polynomial expressions and multi-dimensional arrays to make more code amenable to polyhedral optimization in Polly.

Acknowledgments

The authors would like to thank Florian Sattler for implementing the uniform/affine SCoP classification. This work has been supported by German Research Foundation (DFG) grants LE 912/14-1 and GR 4253/1-1 (project "PolyJIT").

6. REFERENCES

[1] 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)*, pages 377–386. IEEE CS, 2012.

[2] 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)*, volume 6011 of *LNCS*, pages 283–303. Springer, 2010.

[3] 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)*, pages 343–352. ACM, 2010.

[4] 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)*, pages 101–113. ACM, 2008.

[5] C. W. Brown. QEPCAD B: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.*, 37(4):97–108, 2003.

[6] 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*, 14(3):189–204, 2000.

[7] P. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In E. Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 120–133. Springer Berlin Heidelberg, 2004.

[8] M. Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80:233–269, 1973.

[9] J. Doerfert, C. Hammacher, K. Streit, and S. Hack. SPolly: Speculative Optimizations in the Polyhedral Model. In *Proc. Int'l Workshop Polyhedral Compilation Techniques (IMPACT)*, pages 55–61, 2013.

[10] P. Feautrier and C. Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[11] GNU Compiler Collection. `http://gcc.gnu.org/`.

[12] 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*, 34:261–317, 2006.

[13] M. Griebl and C. Lengauer. On the space-time mapping of while-loops. *Parallel Processing Letters*, 4(3):221–232, 1994.

[14] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In *Proc. Int'l Workshop Languages and Compilers for Parallel Computing (LCPC)*, volume 1239 of *LNCS*, pages 603–604. Springer, 1997.

[15] T. Grosser et al. Polly development mailing list. `https://groups.google.com/d/msg/polly-dev/20eI9Xd8Nl0/V6_s2KbOQo4J`, 2012-09-12.

[16] T. Grosser, A. Größlinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012. 28 pp.

[17] 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.

[18] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[19] 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.

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

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

[22] V. Maslov. Delinearization: an efficient way to break multiloop dependence equations. In *In Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 152–161, 1992.

[23] V. Maslov and W. Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. *Parallel Processing: CONPAR 94—VAPP VI*, pages 737–748, 1994.

[24] S. Pop et al. LLVM commits mailing list. `http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20131007/190703.html`, 2013-10-10.

[25] 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)*, pages 1–11. IEEE CS, 2010.

[26] A. Simbürger, S. Apel, A. Größlinger, and C. Lengauer. The Potential of Polyhedral Optimization: An Empirical Study. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Nov. 2013.

[27] 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)*, pages 1–13, 2010. `http://ctuning.org/workshop-grow10`.

[28] S. Verdoolaege. ISL: An integer set library for the polyhedral model. In *ICMS*, pages 299–302. Springer-Verlag, 2010.

[29] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proc. Int'l Conf. Supercomputing (SC)*, pages 262–273. ACM, 2002.

Table 1: Aggregated results of all experiments conducted. The first two groups of columns show static information about the SCoPs found. The third group shows the run time (in %) spent inside uniform (t_u) and affine SCoPs (t_a) of class *Static*. The fourth group shows run-time information for the three classes *Static*, *Algebraic*, and *Dynamic* (in %). The last column shows the lines of LLVM-IR code before any optimizations. A detailled description of all columns can be found in Section 4.3.

Name	S_{st}	S_{pp}	S_{ppa}	S_{md}	S_{na}	S_{jit}	S_{nd}	S_u	S_a	d_u	d_a	t_u	t_a	**Stat**	**Alg**	**Dyn**	Stat | Alg | Dyn	**LoC**
Compilation																		
JS	120	36	95	0	0	150	43	0	77	1	78	0.019	0	0.45	0.47	2.6		910 000
PYTHON	84	6	33	0	0	97	31	0	53	0	87	0	5.3	6.5	6.5	11		730 000
RUBY	100	19	40	0	2	78	44	1	56	3	64	0.0014	0	28	28	31		880 000
TCC	18	0	0	0	0	13	13	2	3	2	4	0	2.4	2.5	2.6	2.6		120 000
Compression																		
7ZA	100	9	24	1	4	65	39	14	51	26	65	0	0	13	14	13		420 000
BZIP2	30	0	0	0	0	5	18	6	6	8	9	1.8	0.44	5.2	5.2	12		29 000
GZIP	23	0	0	0	0	1	15	4	4	4	4	0	0	0.25	0.26	0.49		17 000
XZ	23	1	1	0	0	0	19	1	3	210	4	0	0	0.036	0.038	0.039		77 000
Database																		
LEVELDB	5	0	0	0	0	2	4	0	1	0	2	0	0	0	0	0		73 000
POSTGRES	78	13	0	0	5	84	35	4	49	6	64	0	0	8.5	8.5	8.5		1 100 000
SQLITE3	5	0	0	0	0	0	4	0	1	0	2	0	0	2.0	2.0	2.0		71 000
Encryption																		
BLOWFISH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		3500
BN	24	1	10	0	0	38	8	0	16	0	26	0	22	22	23	36		220 000
CAST	2	0	0	0	0	0	2	0	0	0	0	0	0	25	24	24		3700
CCRYPT	3	0	0	0	0	2	1	0	2	0	1	0	0	1.2	1.2	1.9		14 000
DES	9	0	7	0	0	14	0	0	9	0	11	0	0	0	0	0		47 000
DSA	24	1	10	0	0	38	8	0	16	0	26	0	25	25	25	30		220 000
ECDSA	24	1	10	0	0	38	8	0	16	0	26	0	0.60	0.68	0.68	30		220 000
HMAC	24	1	10	0	0	38	8	0	16	0	26	0	7.5	7.8	8.4	14		220 000
MCRYPT-AES	38	0	0	0	35	33	28	5	4	7	4	0	0	0.72	0.74	1.8		59 000
MCRYPT-CIP	39	0	0	0	37	33	30	5	4	7	4	0	0	0.44	0.44	9.9		61 000
MD5	24	1	10	0	0	38	8	0	16	0	26	0	0	3.9	3.7	9.0		220 000
OPENSSL	67	1	18	0	0	72	23	5	41	6	49	0.13	0	0.13	0.15	0.40		520 000
RC4	1	0	0	0	0	0	1	0	0	0	0	0	0	0.27	0.27	0.27		3300
RSA	24	1	10	0	0	38	8	0	16	0	26	0	22	23	23	45		220 000
SHA1	24	1	10	0	0	38	8	0	16	0	26	0	0.053	0.054	0.059	0.12		220 000
SHA256	24	1	10	0	0	38	8	0	16	0	26	0	0.017	0.025	0.028	0.053		220 000
SHA512	24	1	10	0	0	38	8	0	16	0	26	0	0.053	0.052	0.053	0.100		220 000
SSL	32	1	12	0	0	0	10	2	20	2	29	0	0.15	1.4	1.4	1.4		320 000
Multimedia																		
AVCONV	950	11	310	0	940	1400	0	0	0	0	0	0	0	18	17	45		1 600 000
POVRAY	110	3	71	0	6	53	49	4	53	17	75	1.1	0	8.5	41	42		300 000
X264	55	14	44	0	5	140	17	5	33	9	35	1.3	17	22	25	79		210 000
Scientific																		
LINPACK	9	0	0	0	0	3	2	2	5	8	5	0	45	53	53	53		1400
XEIGTSTC	630	5	5	0	880	430	0	0	0	0	0	0	0	42	44	56		330 000
XEIGTSTD	610	0	0	0	860	380	0	0	0	0	0	0	0	40	40	50		340 000
XEIGTSTS	610	0	0	0	860	380	0	0	0	0	0	0	0	41	41	53		340 000
XLINTSTD	690	0	0	0	1000	630	340	2	380	2	530	0	45	45	46	52		360 000
XLINTSTDS	160	0	0	0	210	120	82	0	74	0	76	0	50	51	52	58		52 000
XLINTSTRFC	170	0	0	0	290	210	72	0	110	0	120	0	23	45	44	54		83 000
XLINTSTRFD	150	0	0	0	230	150	67	0	79	0	81	0	41	43	44	54		64 000
XLINTSTRFS	150	0	0	0	230	150	67	0	79	0	81	0	41	43	43	53		64 000
XLINTSTRFZ	170	0	0	0	290	210	73	0	100	0	110	0	18	37	37	47		82 000
XLINTSTS	530	0	0	0	690	350	250	1	280	1	390	0	41	45	44	51		260 000
XLINTSTZC	190	0	0	0	270	180	90	0	100	0	130	0	51	53	54	58		68 000
Simulation																		
CRAFTY	57	0	9	0	0	2	32	5	20	17	35	0	0	4.0	4.6	8.1		250 000
LAMMPS	330	45	470	3	6	580	190	2	140	11	150	0	0	0	0	0		900 000
LULESH-OMP	16	0	0	0	2	18	16	0	0	0	0	0	0	22	22	23		12 000
LULESH	14	0	0	0	2	17	14	0	0	0	0	0	0	23	23	23		11 000
Verification																		
CROCOPAT	1	0	0	0	0	0	1	0	0	0	0	0	0	3.2	3.2	3.2		89 000
MINISAT	4	0	3	0	0	0	3	0	6	0	6	0	0	17	17	16		22 000