Violated Dependence Analysis

Nicolas Vasilache
Cedric Bastoul

Albert Cohen
Sylvain Girbal

first.last@inria.fr

ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud 11 University, and HiPEAC network

ABSTRACT

The polyhedral model is a powerful framework to reason about high level loop transformations. Yet the lack of scalable algorithms and tools has deterred actors from both academia and industry to put this model to practical use. Indeed, for fundamental complexity reasons, its applicability has long been limited to simple kernels. Recent developments broke some generally accepted ideas about these limitations. In particular, new algorithms made it possible to compute the target code for full SPEC benchmarks while this code generation step was expected not to be scalable.

Instancewise array dependence analysis computes a finite, intensional representation of the (statically unbounded) set of all dynamic dependences. This problem has always been considered non-scalable and/or an overkill with respect to less expressive and faster dependence tests. On the contrary, this article presents experimental evidence of its applicability to full SPEC CPU2000 benchmarks. To make this possible, we revisit the characterization of data dependences, considering relations between time dimensions of the transformed space. Beyond algorithmic benefits, this naturally leads to a novel way of reasoning about violated dependences across arbitrary transformation sequences. Reasoning about violated dependences relieves the compiler designer from the cumbersome task of implementing specific legality checks for each single transformation. It also allows, in the case of invalid transformations, to precisely determine the violated dependences that need to be corrected. Identifying these violations can in turn enable automatic correction schemes to fix an illegal transformation sequence with minimal changes.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Algorithms, Languages, Theory, Performance

Keywords

Static Analysis, Optimizing and Parallelizing Compilers

ICS06, June 28–30, Cairns, Queensland, Australia.

1. INTRODUCTION

The power of an automatic optimizer or parallelizer greatly depends on its capacity to decide whether two portions of the program execution may be interchanged or run in parallel. Such knowledge is related to the difficult task of *dependence analysis* which aims at precisely disambiguating memory references.

A number of works proposed data dependence tests and abstractions, with various motivations such as computational cost, precision or application domain (see Section 2 for details and useful references). Several empirical studies have been conducted to compare those tests [29, 18, 30, 31]. The generally accepted conclusion is: "it is more interesting to use simple tests (like the *Banerjee-test* [40] or *I-test* [22]), and simple abstractions (like *Direction Vectors* [39]), because they capture most data dependence information at a low computational cost". The present paper contrasts with such generally accepted ideas and with the traditional use of data dependence analysis.

First we recall how, using an exact instancewise analysis, a dependence between two statements does not necessarily hamper the application of an optimizing/parallelizing transformation. Indeed, the comparison of data dependence tests in above-mentioned studies is quite biased: it only evaluates the ability to prove or disprove dependences between statements, and not to precisely tell which iterations of those statements are in dependence. The essence of data dependence analysis is to build or to check useful transformations. To prove or disprove dependences between statements is quite rough since optimization/parallelization may often be possible even if there exists data dependences. On the contrary, instancewise analyses and abstractions give the right precision level to decide wether or not to apply a loop transformation. Empirical studies of dependence tests/analyses which ignore the impact on transformations are not powerful enough for advanced compiler design.

Second we show that beyond dependence analysis, an illegal transformation is not necessarily a dead-end. We will show how to exactly determine the violated dependences that need to be corrected. Identifying these violations can in turn enable automatic correction schemes to fix an illegal transformation sequence with minimal changes.

Third, we provide empirical evidence of the scalability of an exact instancewise dependence analysis. We present algorithmic contributions which allowed to present the first experimental validation of instancewise analysis on full SPEC CPU2000 benchmarks.

The paper is organized as follows: Section 2 revisits the vast amount of related works to motivate our approach; Section 3 gives an original characterization of instancewise dependences, then describes the key algorithmic improvements for scalability; Section 4 details the computation of sets of violated dependences after a given transformation sequence, then presents a fast and accurate legality test based on the affine form of the Farkas lemma; Section 5 presents experimental results; and Section 6 explores the potential of automatic correction approaches on a realistic example.

2. RELATED WORKS

Many tests have been designed for dependence checking between different statements or between different executions of the same statement. It has been extensively shown that this problem amounts to detecting whether or not a system of equations has an integer solution inside a region of $\mathbb{Z}^n$ [4].

Most of the dependence tests try to find efficiently a reliable, approximative but conservative (they overestimate data dependences) solutions. The GCD-test [3] has been the very first practical solution, it is still present in many implementations as a first check with low computational cost. This test assumes that if the greatest common divisor of the coefficients of an equation divides the constant term, then a solution exists. A generalized GCD-test has been proposed to handle multi-dimensional array references [4]. The Banerjee test uses the intermediate value theorem to disprove a dependence: it computes the upper and lower bounds of an equation and checks if the constant part lies in that range [40]. The λ-test is an extension to this test that handles multi-dimensional array references [22]. Some other important solutions are a combination of GCD and Banerjee tests called I-test [22], the Δ-test [18] that gives an exact solution when there is at most one variable in the subscript functions, and the Power-test which uses the Fourier-Motzkin variable elimination method [36] to prove or disprove dependences [41]. Beside their approximative nature, these dependence tests suffer from many other major limitations. The most stringent one is their inability to precisely handle `if` conditionals, loops with parametric bounds, triangular loops (a loop bound depends on an outer loop counter), coupled subscripts (two different array subscripts refer the same loop counter), or parametric subscripts.

On the opposite, a few methods allow to find an exact solution to the dependence problem, but at a higher computational cost. The Omega-test is an extension to the Fourier-Motzkin variable elimination method to find integral solutions [32]. On one hand, once a variable is eliminated, the original system has an integer solution only if the new system has an integer solution (if this is not the case there is no solution). On the other hand, if an integer point exists in a space computed from the new system, then there exists an integer point in the original system (if this is the case, there is a solution). The PIP-test uses a parametric version of the dual-simplex method with Gomory cuts to find an integral solution [15]. These two tests not only give an exact answer, they are also able to deal with complex loop structures and (affine) array subscripts. The PIP-test is more precise than the Omega-test when dealing with parametric codes (when one or more integer symbolic constant are present), for instance, in the following pseudo-code:

```
for(i=0; i<=N; i++) {
  A[i] = ...;
  ... = ... A[i+100] ...;
}
```

the Omega-test will state that there is a dependence between the two statements while the PIP-test will precise that the dependence only exists if N is greater or equal to 100. Both tests have worst-case exponential complexities but work quite well in practice as shown by Pugh for the Omega-test [32]. Other costly exact tests exist in the literature [28, 12] but are often not able to handle complex control in spite of their cost.

We do not advocate for the use of any of these tests, but rather for the computation of *instancewise* dependence information as precisely as possible, i.e., for intensionally describing the statically unbounded set of all pairs of dependent statement iterations, called *instances*. Dependence tests are statementwise decision problems associated with the existence of a pair of dependent instances, while instancewise dependence analysis provides additional information that can enable finer program transformations, like affine scheduling [23, 16, 25, 19]. The intensional characterization of instancewise dependences can take the form of multiple *dependence abstractions*, depending on the precision of the analysis and on the requirements of the user. The simplest and least precise one is called *dependence levels*, it specifies for a given loop nest which loop carry the dependence. It has been introduced in the Allen and Kennedy parallelization algorithm [1]. The *direction vectors* is a more precise abstraction where the i-th element approximates the value of all the i-th elements of the *distance vectors* (which shows the difference of the loop counters of two dependent instances). It has been introduced by Lamport [23] and formalized by Wolfe [39] and is clearly the most widely used representation. The most precise abstraction is the *dependence polyhedron* [20] which is able to determine exactly the set of statement instances in dependence relation. The choice of a given dependence abstraction is crucial for further study: choosing an imprecise one can result in blacking out interesting transformations. For instance, let us consider the following example:

```
  for(i=0; i<=N; i++)
    for(j=0; j<=N; j++)
S     A[i][j] = A[j][i] + A[i][j-1];
```

There are three dependences in this loop nest (a read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,i,j+1\rangle$, another read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,j,i\rangle$ and a write-after-read from $\langle S,j,i\rangle$ to $\langle S,i,j\rangle$). Dependence levels are 2, 1 and 1: each loop carries at least one dependence and no parallelism can be found. Direction vectors are $(0,1)$, $(+,-)$, $(+,-)$: the second coefficients 1 and $-$ hamper any parallelism detection. Using dependence polyhedra, parallelism may be found: the Feautrier algorithm suggests the affine schedule $\theta(i,j) = 2i + j - 3$ (all instances with the same schedule may be run in parallel), see [38]. In the rest of this paper, we compute the most precise representation of dependences: dependence polyhedra.

3. POLYHEDRAL DEPENDENCES

A thorough tool to perform dependence analysis on loop nests is the polytope model. The dependence analysis is then deemed exact. We use the notations of the URUK framework [9] briefly restated in Section 3.1; under the invariants of this framework, it is possible to apply any transformation at a given point without worrying about its validity. We therefore lift the tedious constraint of ensuring the legality of a transformation before being able to apply it. Only after a full transformation sequence has been applied do we care about the correctness of the sequence as a whole.

3.1 Normalized Polyhedral Representation

This section is a quick overview of our polyhedral framework. It is based on a normalized representation of programs and transformations, derived from the models introduced by Pugh and Feautrier [33, 16].

We will use the following vocabulary and notations. The scope of all program manipulations is a sequence of loop nests with constant strides and affine bounds. It includes non-rectangular, non-perfectly nested loops, and conditionals with boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to a representation in the polyhedral model. We call *Static Control Part* (SCoP) any *maximal syntactic program segment* satisfying these constraints. All variables that are invariant within a SCoP are called *global parameters*. For each statement within a SCoP, the representation separates three attributes, characterized by parametric matrices: the iteration domain, the schedule, and the access functions. Even though transformations can still be applied to loops or full procedures, they are individually applied to each statement for a maximal flexibility.

Given a statement S within a SCoP, let d^S be the depth of S, let $\mathbf{i}$ be the *iteration vector* of S, i.e., the vector of loop indices to which S belongs (the dimension of $\mathbf{i}$ is d^S), and let $\mathbf{g}$ be the vector of $d_\mathbf{g}$ *global parameters* and the scalar component. The *iteration domain* of S is defined by

$$\left\{\mathbf{i} \mid D^S(\mathbf{i}|\mathbf{g})^t \geq \mathbf{0}\right\},$$

where D^S is the matrix of n affine constraints on the execution of statement S; D^S has n rows and $d^S + d_\mathbf{g} + 1$ columns. Each iteration $\mathbf{i}$ of a statement S is called an *instance*, and is denoted by $\langle S, \mathbf{i}\rangle$. The *schedule* of S is an affine function mapping iterations of S to multidimensional time-stamps. This function is encoded by a matrix Θ^S of $2d^S + 1$ rows and $d^S + d_\mathbf{g} + 1$ columns. By definition, the execution order of all iterations of all statements is the lexicographic order on multi-dimensional time-stamps of the schedules of these statements.

Each reference in a statement S is of the form $\langle \mathtt{X}, f(\mathbf{i}^S)\rangle$ where X is an array and f is the subscript function. When it is affine, this function is defined by

$$f(\mathbf{i}^S) = \left(\mathrm{Acc}_\mathbf{i}|\mathrm{Acc}_\mathbf{g}\right)\mathbf{i}^S,$$

where $\mathrm{Acc}_\mathbf{i}$ is a matrix with as many lines as array dimensions in X, resp. $\mathrm{Acc}_\mathbf{g}$ for the global parameters and the scalar component. We also consider more general non-affine references with conservative approximations.

Polyhedral compilation usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code. This paper focuses on computing instancewise dependence information effectively to improve the applicability and expressiveness of the transformation step.

It is well known that arbitrarily complex sequences of loop transformations can be captured in one single transformation step of the polyhedral model. This was best illustrated by affine scheduling [21] and partitioning [25] algorithms. Yet to ease the composition of program transformations on the polyhedral representation, we further split the representation of the schedule into smaller matrix and vector blocks satisfying strong normalization rules:

$$\Theta^S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ A_{1,1}^S & \cdots & A_{1,d^S}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,d_\mathbf{g}}^S & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ A_{2,1}^S & \cdots & A_{2,d^S}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,d_\mathbf{g}}^S & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{d^S,1}^S & \cdots & A_{d^S,d^S}^S & \Gamma_{d^S,1}^S & \cdots & \Gamma_{d^S,d_\mathbf{g}}^S & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_{d^S}^S \end{array}\right],$$

where A^S is a square matrix mapping iterations vectors to odd time dimensions (time dimension start at depth 0), where the Γ^S matrix allows to shift the schedule with respect to global parameters, and where β^S scatters the statements in the multidimensional time in mapping every statement to a specific vector of even time dimensions. Such encodings with $2d^S + 1$ dimensions were previously proposed by Feautrier, then by Kelly and Pugh [21]. The URUK framework adds a few additional normalization rules, formally defined in [17]; these rules have been left out of this quick overview since they were not directly useful to the dependence analysis itself.

3.2 Instancewise Polyhedral Dependences

The purpose of this paper is to compute the exact dependence information between every pair of instances, i.e., every pair of statement iterations. Considering a pair of statements S and T of where at least one is a write, there is a dependence from an instance $\langle S, \mathbf{i}^S\rangle$ of S to an instance $\langle \mathbf{i}^T, T\rangle$ of T (or $\langle T, \mathbf{i}^T\rangle$ depends on $\langle S, \mathbf{i}^S\rangle$) if and only if the following *instancewise* conditions are met:

Execution condition: both instances belong to the corresponding statement iteration domain: $D_\mathbf{i}^S\mathbf{i}^S \geq 0$ and $D_\mathbf{i}^T\mathbf{i}^T \geq 0$,

Conflict condition: both instances refer the same memory location: $\left(\mathrm{Acc}_\mathbf{i}^S|\mathrm{Acc}_\mathbf{g}^S\right)\mathbf{i}^S = \left(\mathrm{Acc}_\mathbf{i}^T|\mathrm{Acc}_\mathbf{g}^T\right)\mathbf{i}^T$, and

Causality condition: the instance $\langle S, \mathbf{i}^S\rangle$ is executed before $\langle T, \mathbf{i}^T\rangle$ in the original execution: $\Theta^S\mathbf{i}^S \ll \Theta^T\mathbf{i}^T$,

where $\ll$ denotes the lexicographic order on vectors.

As reminded in Section 3.1, the schedule (the multidimensional time-stamp) at which an instance is executed is determined, for statement S, by the $2d^S + 1$ vector given by $\Theta^S\mathbf{i}^S$. Relative order between instances is given by the relative lexicographic order of their schedule vectors.

Consider the original polyhedral representation of a program, before any transformation has been applied. For a given statement S, matrix A^S is the identity, Γ^S is 0 and vector β^S captures the syntactic position of S in the original code. In this configuration, the three aforementioned conditions correspond to the classical definition of polyhedral dependences [13, 32].

A dependence is said to be loop independent if the causality condition $\Theta^S\mathbf{i}^S \ll \Theta^T\mathbf{i}^T$ is resolved sequentially on one of the β components of the schedule. A dependence is said to be loop carried at loop depth p if the causality condition is resolved by the $\left(\mathrm{Acc}_\mathbf{i}^S|\mathrm{Acc}_\mathbf{g}^S\right)$ component of the schedule at depth p:

Loop-carried dependence at depth p:

$$\beta_{0..p-1}^S = \beta_{0..p-1}^T, \left((A^S|\Gamma^S)\mathbf{i}^S\right)_{0..p-1} = \left((A^T|\Gamma^T)\mathbf{i}^T\right)_{0..p-1},$$
$$\text{and } \left((A^S|\Gamma^S)\mathbf{i}^S\right)_p < \left((A^T|\Gamma^T)\mathbf{i}^T\right)_p.$$

Loop-independent dependence at depth p:

$$\beta_{0..p-1}^S = \beta_{0..p-1}^T,$$
$$\left((A^S|\Gamma^S)\mathbf{i}^S\right)_{0..p-1} = \left((A^T|\Gamma^T)\mathbf{i}^T\right)_{0..p-1} \text{ and } \beta_p^S < \beta_p^T.$$

The purpose of dependence analysis is to compute a directed dependence multi-graph DG_p for each possible dependence level p. Unlike traditional reduced dependence graphs, an arc ST in DG_p is labeled by a dependence polyhedron capturing the set of pairs of iteration vectors ($\mathbf{i}^S$, $\mathbf{i}^T$) at the desired dependence level. These pairs of belong to the Cartesian product space $\mathrm{P}^{\mathrm{S,T}}$ of dimension $(d^S + d_\mathbf{g} + 1) + (d^T + d_\mathbf{g} + 1)$ and meet the instancewise dependence conditions. Since the global parameters are invariant across the whole SCoP, we can remove redundant parameter dimensions and project this space into the equally expressive one of dimension $d^S + d^T + d_\mathbf{g} + 1$. We will indistinctly use $\mathbf{i}^S$ to represent an iteration vector for statement S in either the original space of dimension

$d^S + d_{\mathbf{g}} + 1$ or in one of the two aforementioned Cartesian product spaces. It will be clear from the context which one should be assumed.

3.3 Fast Dependence Analysis

The decomposition of schedule matrices into the A, β and Γ components allows to break down the Cartesian product space into smaller blocks (the $\bullet$ symbol denotes *all rows*):

$$D^S_{\mathbf{i}} = D^S_{\bullet,1..d^S} \qquad D^S_{\mathbf{g}} = D^S_{\bullet,d^S+1..d^S+d_{\mathbf{g}}+1}$$
$$\mathrm{Acc}^S_{\mathbf{i}} = \mathrm{Acc}^S_{\bullet,1..d^S} \qquad \mathrm{Acc}^S_{\mathbf{g}} = \mathrm{Acc}^S_{\bullet,d^S+1..d^S+d_{\mathbf{g}}+1}$$

For conciseness reasons, we will use the terminology *dependence of depth* $-p$ rather than loop-independent dependence of depth p, and the terminology *dependence of depth* p rather than loop-carried dependence of depth p. Then, for a dependence from S to T of depth level p, we have

$$\beta^S_{0..|p|-1} = \beta^T_{0..|p|-1}.$$

If $p < 0$, the causality condition is enforced by

$$\beta^S_{|p|} < \beta^T_{|p|}$$

otherwise it is enforced by:

$$\left(A^S_{\bullet,p}|\Gamma_{\bullet,p}\right)\mathbf{i}^S < \left(A^T_{\bullet,p}|\Gamma_{\bullet,p}\right)\mathbf{i}^T.$$

In the dependence graph, there will be an arc from S to T iff for the considered depth, the beta relations are enforced and if the dependence polyhedron of Figure 2 (if $p \le 0$) or Figure 1 (if $p > 0$) is not empty. All dependence polyhedra for a pair of statements and a given pair of access functions are disjoint by construction.

$$\left(\begin{array}{c|c|c}
D^S_{\mathbf{i}} & 0 & D^S_{\mathbf{g}} \ge 0 \\ \hline
0 & D^T_{\mathbf{i}} & D^T_{\mathbf{g}} \ge 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & -\mathrm{Acc}^T_{\mathbf{i}} & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^T_{\mathbf{g}} = 0 \\ \hline
-A^S_{\bullet,1..p-1} & A^T_{\bullet,1..p-1} & -\Gamma^S_{\bullet,1..p-1} + \Gamma^T_{\bullet,1..p-1} = 0 \\ \hline
-A^S_{\bullet,p} & A^T_{\bullet,p} & -\Gamma^S_{\bullet,p} + \Gamma^T_{\bullet,p} \ge 1
\end{array}\right)$$

Figure 1: Dependence at depth $p > 0$

$$\left(\begin{array}{c|c|c}
D^S_{\mathbf{i}} & 0 & D^S_{\mathbf{g}} \ge 0 \\ \hline
0 & D^T_{\mathbf{i}} & D^T_{\mathbf{g}} \ge 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & -\mathrm{Acc}^T_{\mathbf{i}} & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^T_{\mathbf{g}} = 0 \\ \hline
-A^S_{\bullet,1..p} & A^T_{\bullet,1..p} & -\Gamma^S_{\bullet,1..p} + \Gamma^T_{\bullet,1..p} = 0
\end{array}\right)$$

Figure 2: Dependence at depth $p \le 0$

This decomposition dramatically reduces the cost of computing the polyhedron of all pairs of dependence instances spawned by statements S and T at a given depth p. Indeed, a simple lookup on β vectors can quickly eliminate costly polyhedral operations when the prerequisite on β prefixes of dimension $p-1$ is not met: $\beta^S_{1..p-1} = \beta^T_{1..p-1}$. For a dependence at depth $p \le 0$ the constraint system is the same except for the last row, as explained previously.

Special cases arise when examining scalar references or arrays indexed by nonlinear access functions. The latter are conservatively dealt with as scalars by not adding access constraints from the dependence polyhedron. In such conservative cases, all source and sink statements are considered to access the same memory location and are bound to respect the causality order after transformation. There exist of course finer approximation schemes, but since algorithmic complexity is higher, their practical applicability to large programs remains unknown [10, 42, 5].

Let l be the first index such that $\beta^S_l < \beta^T_l$; the only possible loop independent dependence is of depth $-l$. The reader may notice the great similarities between dependence polyhedra on consecutive dependence levels. Exploiting these similarities allowed us to design an efficient algorithm that traverses the dependence levels in increasing absolute value order (0, 1, -1, ..., l, $-l$). At depth level 0, all but the causality constraints are combined in the dependence polyhedron which is added to the dependence graph DG_0 and saved into a cache. At depth level 1, the polyhedron is recovered from the cache of depth 0 and intersected with the constraint

$$\left(A^S_{\bullet,1}|\Gamma_{\bullet,1}\right)\mathbf{i}^S < \left(A^T_{\bullet,1}|\Gamma_{\bullet,1}\right)\mathbf{i}^T.$$

The result, if non-empty, is added to DG_1. At depth level -1, the polyhedron is recovered from the cache of depth 0 and intersected with the constraint

$$\left(A^S_{\bullet,1}|\Gamma_{\bullet,1}\right)\mathbf{i}^S = \left(A^T_{\bullet,1}|\Gamma_{\bullet,1}\right)\mathbf{i}^T.$$

If the resulting polyhedron is empty, adding more constraints is useless and translates immediately into a short-circuit of the computations. On the other hand, if the result is not empty, it is added to the cache for later reuse. This short-circuiting strategy combined to the fact that constraints can be added one at a time greatly reduces the overall computation time in the context of costly polyhedral operations that have exponential complexity.

We have shown how to efficiently compute a dependence polyhedron for a given pair of statements on a given pair of accesses to the same array. Technically, the polyhedral operations involved can be implemented with the PolyLib [26], a state-of-the-art library to operate on parameterized $\mathbb{Z}$-polyhedra.

3.4 Transitively-Covered Dependences

In general, the full dependence graph contains redundant information associated with transitively covered dependences. This incurs computational overhead in subsequent optimization, scheduling or legality checking phases.

The standard technique to eliminate redundant information consists in removing all memory-based dependences by converting the SCoP to (dynamic) single assignment. This transformation amounts to array renaming and expansion (a generalization of array privatization), using the array data-flow analysis technique proposed by Feautrier [13, 15]. This method only considers flow dependences and computes for each statement and each reference it *reads* the last producer of the value read. The algorithm walks the code backwards, calling the PIP library [14] to incrementally build the result. The solution is a quasi affine selection tree (generalization of a “last write tree” [27]) implementing the case distinctions for pertinent values of the target statement’s iterators and invariant parameters associated with distinct producers (or distinct affine forms). As a dependence graph compression, the major drawback of this approach is the need to operate on a single-assignment program, hence to resort to complex array contraction and storage mapping optimization techniques to ultimately reduce the memory footprint [24, 34].

Our method does not require conversion to single assignment form. Instead, for each target instance, we do identify the last source of a (dynamic) dependence targetting this precise instance, in order to remove transitively covered dependences, but we consider all dependences including the memory based anti (write-after-read) and output (write-after-write) ones. The result is a simplified dependence graph for each depth level, bearing the exact simplified dependence relation. Consider a dependence from S to T of depth p on a given memory location x. The key to our approach is to determine, which are all the possible statements that can be interleaved between the time of execution of a source iteration and the time of execution of its corresponding target iteration(s) (possibly many). We consider a candidate covering statement C that *writes* to x and reason in the Cartesian product of the three former spaces, which in turn may be collapsed into the smaller space $\mathrm{P}^{\mathrm{S,C,T}}$ of dimension $d^S + d^C + d^T + d_{\mathbf{g}} + 1$. C must satisfy the following conditions:

- it must be the target of a dependence of depth p from S;
- it must be the source of a dependence of depth p to T.

The layout of the covering polyhedron corresponds to Figure 3 if $p > 0$ or to Figure 4 if $p \leq 0$.

$$\left(\begin{array}{c|c|c|c}
D^S_{\mathbf{i}} & 0 & 0 & D^S_{\mathbf{g}} \geq 0 \\ \hline
0 & D^C_{\mathbf{i}} & 0 & D^C_{\mathbf{g}} \geq 0 \\ \hline
0 & 0 & D^T_{\mathbf{i}} & D^T_{\mathbf{g}} \geq 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & -\mathrm{Acc}^C_{\mathbf{i}} & 0 & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^C_{\mathbf{g}} = 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & 0 & -\mathrm{Acc}^T_{\mathbf{i}} & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^T_{\mathbf{g}} = 0 \\ \hline
-A^S_{\bullet,1..p-1} & A^C_{\bullet,1..p-1} & 0 & -\Gamma^S_{\bullet,1..p-1} + \Gamma^C_{\bullet,1..p-1} = 0 \\ \hline
-A^S_{\bullet,1..p-1} & 0 & A^T_{\bullet,1..p-1} & -\Gamma^S_{\bullet,1..p-1} + \Gamma^T_{\bullet,1..p-1} = 0 \\ \hline
-A^S_{\bullet,p} & A^C_{\bullet,p} & 0 & -\Gamma^S_{\bullet,p} + \Gamma^C_{\bullet,p} \geq 1 \\ \hline
0 & -A^C_{\bullet,p} & A^T_{\bullet,p} & -\Gamma^C_{\bullet,p} + \Gamma^T_{\bullet,p} \geq 1
\end{array}\right)$$

Figure 3: Covering polyhedron at depth $p > 0$

$$\left(\begin{array}{c|c|c|c}
D^S_{\mathbf{i}} & 0 & 0 & D^S_{\mathbf{g}} \geq 0 \\ \hline
0 & D^C_{\mathbf{i}} & 0 & D^C_{\mathbf{g}} \geq 0 \\ \hline
0 & 0 & D^T_{\mathbf{i}} & D^T_{\mathbf{g}} \geq 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & -\mathrm{Acc}^C_{\mathbf{i}} & 0 & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^C_{\mathbf{g}} = 0 \\ \hline
\mathrm{Acc}^S_{\mathbf{i}} & 0 & -\mathrm{Acc}^T_{\mathbf{i}} & \mathrm{Acc}^S_{\mathbf{g}} - \mathrm{Acc}^T_{\mathbf{g}} = 0 \\ \hline
-A^S_{\bullet,1..p} & A^C_{\bullet,1..p} & 0 & -\Gamma^S_{\bullet,1..p} + \Gamma^C_{\bullet,1..p} = 0 \\ \hline
-A^S_{\bullet,1..p} & 0 & A^T_{\bullet,1..p} & -\Gamma^S_{\bullet,1..p} + \Gamma^T_{\bullet,1..p} = 0
\end{array}\right)$$

Figure 4: Covering polyhedron at depth $p \leq 0$

Due to the transitivity of the equality and inequality relations, access and schedule relations between S and T can be omitted, contributing to lowering the computational cost of the covering polyhedron. The constraint on β have been overlooked, they are nonetheless the first step in the selection of a candidate covering statement C: for a dependence of depth $p \leq 0$ the necessary constraint on β is $\beta^S_{0..p-1} = \beta^C_{0..p-1} = \beta^T_{0..p-1}$ and $\beta^S_p < \beta^C_p < \beta^T_p$. For a dependence of depth $p > 0$, the constraint reduces to $\beta^S_{0..p-1} = \beta^C_{0..p-1} = \beta^T_{0..p-1}$.

Consider the example in Figure 5 where three statements assign array `X` with affine access functions. Suppose there is an output dependence from S to T at depth 1. It is not trivial to see, on the syntactic code, that C may actually cover part of this dependence. However, a quick look at the β vectors, $\beta^S = (0,0,0)^t$, $\beta^T = (0,0,1)^t$, $\beta^C = (0,1)^t$, show they satisfy the necessary constraint for $p = 1$. Should C really shadow a subset of the dependence polyhedron is determined by the non-emptiness of the constraint set of Figure 3.

```
  for(i=M; i<=N; i++) {
    for(j=P; j<=Q; j++) {
S     X[f(i,j)] = ...;
T     X[g(i,j)] = ...;
    }
C   X[h(i)] = ...;
  }
```

Figure 5: Covering statement example

Once a covering polyhedron has been computed and deemed not empty, it is necessary to relate it to the points in $\mathrm{P}^{\mathrm{S,T}}$ that it shadows. The shadowed polyhedron is the projection of the covering polyhedron on $\mathrm{P}^{\mathrm{S,T}}$. The projection being a standard PolyLib operation, we will not get into further details here. The last step is then to remove the shadowed polyhedron from the original dependence polyhedron. Once again this is a standard PolyLib operation that may however return a non-convex list of convex polyhedra.

Finally, special care must be taken when dealing with conservative dependence approximations associated with non-affine array subscripts. Our current implementation preserves every transitively covered dependence arc when the source or target of this arc is a non-affine reference. More precise methods have been proposed but their practical evaluation is left for future work [42, 5].

4. TRANSFORMATION LEGALITY

In general, loop optimizers apply legality checks before transforming the program. More precisely, every loop transformation is associated with specific legality conditions, and sometimes specific static analyses [2]. For instance, the unimodular transformations on one side, and the loop fusion/fission on the other side, require distinct legality checking code. This traditional approach has several drawbacks:

- it is almost impossible to define complex loop transformations with a global impact on the loop nest, since their legality conditions would be difficult to formally define [21];
- hence complex transformations must be decomposed into sequences of primitive ones, a fragile and combinatorial task in general [9];
- each individual transformation must be checked, leading to compile-time overhead and additional fragility, since a single conservative approximation for one of these checks may invalidate the whole sequence [17];
- in terms of compiler engineering, more effort is needed to scatter and specialize legality checking code in the loop transformation infrastructure [2].

In addition, the ability to check transformations after they have been applied enables new ways to drive an optimization process: if the compiler can reason about violated dependences, some fundamental decision flaws of syntactic compilers disappear *by converting early decisions into delayed corrections of illegal transformations*. For example, a typical ill-formed optimization problem like "is there a loop peeling step that would enable fusion of two given loops ?" would simply be converted into the extraction of the minimal set of iterations that violate the fusion, followed by the natural peeling transformation to correct this violation.

In the context of these fundamental and compiler engineering motivations, this section explains how instancewise dependence information can be used to delay legality checks *after* the application of complex transformations or long transformation sequences.

4.1 Characterization of Violated Dependences

After transforming the SCoP, the question arises whether the resulting program still executes correct code. Our approach consists in saving dependence graphs at each depth, before applying any transformation, then to apply a given transformation sequence, and eventually to run a legality check at the very end of the sequence.

We shall denote the *transformed* matrices, polyhedra, statements and depth by the same letters as before with the addition of a *prime* symbol. For instance, a transformed statement S' with transformed schedule components $A^{S'}$, $\beta^{S'}$ and $\Gamma^{S'}$ corresponds to original statement S with schedule components A^{S}, β^{S} and Γ^{S}. Again we consider a dependence from S to T in the original code and we want to determine if it has been preserved in the transformed program. For the moment we shall consider transformations that only modify the schedule. For such schedule transformations, the mapping $S \rightarrow S'$ is a bijection. Whether or not this translates into statement duplication in the resulting code is determined by the code generator [35, 6].

The violated dependences analyzer computes the iterations of the Cartesian product space $\mathrm{P}^{S',T'}$ (which is isomorphic to $\mathrm{P}^{S,T}$) that were in a dependence relation in the original program and *whose order has been reversed by the transformation*. These iterations, should they exist, do not preserve the causality of the original program. By reasoning in the transformed space, it is straightforward to see that the set of iterations that violate the causality condition is the intersection of a dependence polyhedron with the constraint set $\Theta^{S'}\mathbf{i}^{S'} \geq \Theta^{T'}\mathbf{i}^{T'}$. This in turn translates into the same case distinction as in Section 3.2. Considering a dependence polyhedron from S to T at depth p denoted by $\delta_p^{S\rightarrow T}$, we are looking for the exact set of iterations of $\delta_p^{S\rightarrow T}$ such that there is a dependence from T' to S' at transformed depth p'. This gives rise to the case distinction of Figure 6 if $p' > 0$, and of Figure 7 if $p' \leq 0$. Note that for the case $p' \leq 0$, the violated dependence is actually the set of iterations that are *potentially* in violation. The additional constraint $\beta^{S'}_{|p'|} > \beta^{T'}_{|p'|}$ is also needed.

$$\left(\begin{array}{c|c|c}
\multicolumn{3}{c}{\delta_p^{S\rightarrow T}} \\
\hline
-A^{S'}_{\bullet,1..p'-1} & A^{T'}_{\bullet,1..p'-1} & -\Gamma^{S'}_{\bullet,1..p'-1}+\Gamma^{T'}_{\bullet,1..p'-1}=0 \\
\hline
A^{S'}_{\bullet,p'} & -A^{T'}_{\bullet,p'} & \Gamma^{S'}_{\bullet,p'}-\Gamma^{T'}_{\bullet,p'}\geq 1
\end{array}\right)$$

Figure 6: Violated dependence at depth $p' > 0$

$$\left(\begin{array}{c|c|c}
\multicolumn{3}{c}{\delta_p^{S\rightarrow T}} \\
\hline
A^{S'}_{\bullet,1..p'} & -A^{T'}_{\bullet,1..p'} & \Gamma^{S'}_{\bullet,1..p'}-\Gamma^{T'}_{\bullet,1..p'}=0
\end{array}\right)$$

Figure 7: Violated dependence candidates at depth $p' \leq 0$

A violated dependence polyhedron, as defined in Figure 6, will be referred to as $\delta_{\mathrm{vio}}{}_{p'}^{S'\rightarrow T'}$. The prerequisites on $\beta'_{1..|p'|-1}$ are the same as in Section 3.2 since we are essentially solving the *same* problem in the transformed space. Our algorithm performs the same incremental constraint intersection as before, and the same caching mechanism is used to speed up computations. At each depth p, the resulting non empty polyhedra will be gathered into a *violated dependence graph*, denoted by $\mathrm{VDG}_{p'}$. Beyond characterizing illegal transformation sequences, it is possible to reason about these graphs of violated dependences and effectively derive more flexible optimization algorithms. Further details will come later in the paper.

Finally, to avoid enforcing unnecessary constraints in reductions or scans [2], it is also possible to consider fundamental properties such as commutativity and associativity, to further refine the violated dependence graph.

4.2 Tracking Domain Transformations

So far we have only considered schedule transformations which consist in reordering the iterations of statements according to affine parametric scheduling functions. These transformations encompass fusion, fission, shifting, loop reversal, interchange, skewing and unimodular transformations. It is also important to note that the dependence analysis supports *non-unimodular* but also *non-invertible* schedule transformations (i.e. for which the A matrix is singular). However, some transformations cannot be expressed as schedule transformations only and special care must be taken for these. Strip-mining, which is an ingredient of loop tiling [39], consists of expanding the dimension of statement and adding new constraints to the expanded domain matrix. Peeling and index-set splitting, on the other hand, break a domain into disjoint parts, each having its own different schedule [9]. It is then necessary to duplicate the statements in our formalism, each new statement bearing its own domain constraint. Note that, whatever the length of the final generated code, peeling and index-set splitting are the *only transformations* that create new statements. (Loop unrolling is treated as strip-mining with a specific code generation flag [37], it does not impact the size of the polyhedral representation.) The intrinsic complexity of the transformation will be handled by the code generator, for which we previously demonstrated good scalability properties [37].

The ability to seamlessly integrate non-invertible transformations and domain transformations in our delayed dependence checking approach is a strong contribution to compiler robustness and scalability. The solution is to carry locally, along with each statement, a history of domain transformations in the order they have been applied. For illustration purposes, we will consider a dependence $\delta_p^{S\rightarrow T}$ from S to T, some unspecified schedule transformations, and the following domain transformations interleaved among the schedule transformations:

- S is strip-mined along first time dimension by a factor of 4; its domain is extended by the inequalities associated with the striped iteration blocks (domain matrix SM^S decomposed into $\mathrm{SM}^S_{\mathbf{i}}$ and $\mathrm{SM}^S_{\mathbf{g}}$ as in Sec. 3.2);

- T is split according to the partitioning conditions C_{ond} and $\neg C_{ond}$ into T'_1 and T'_2 respectively.

The mapping $T \rightarrow T'$ becomes a one-to-many mapping in the wake of domain transformations. When checking legality for dependence $\delta_p^{S\rightarrow T}$ at depth 0, the analyzer first replays the domain transformations encoded in the history, then saves the result in cache before starting to add any schedule constraint. This opportunity of decoupling the domain transformations from the schedule transformations is a strong property of the URUK framework [17]. Strip-mining is an exception: it couples domain and schedule transformations. It is always legal, but the only way to make it compositional with other schedule transformations is to apply it to *time dimensions* of the transformed space [17]. The domain constraints created by a strip-mine then bear the current schedule information at the time of the strip-mine.

Back to our example, we show the construction of violated dependence polyhedra $\delta_{\text{vio}_{p'}}^{S'\to T'_1}$ and $\delta_{\text{vio}_{p'}}^{S'\to T'_2}$ only for depth T'_1 and $p' > 0$ in Figure 8 (the matrix T'_2 can be obtained in negating C_{ond} in T'_1). [1]

4.3 Fast Legality Check

The previous characterization allows to compute exact sets of violated dependences. This is an overkill for legality checking. This section describes a fast dependence test that may be applied *after* the application of arbitrarily complex program transformations. This test can also be used to quickly filter out satisfied dependences when computing violated dependence polyhedra.

In practice, the problem of Figure 6 can be checked efficiently without using costly polyhedral operations. Consider a transformed dependency depth $p' > 0$. The left-hand side of the inequality in the last row represents the violation amount on depth p', i.e., the amount of time by which S' is late with respect to T' in the transformed schedule; we shall denote it by $\Delta_{p'}^{S'\to T'}$. The problem is to determine whether or not, $\Delta_{p'}^{S'\to T'}$ can be positive on $\mathcal{P}_{p'-1}$, the violated dependence candidates polyhedron at depth $p'-1$ defined by the two first rows of Figure 6. This is solved by application of the affine form of the Farkas Lemma [16, 36].

LEMMA 1 (AFFINE FORM OF FARKAS LEMMA). *Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\mathbf{x}+\mathbf{b} \geq 0$. Any affine function $f(\mathbf{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

$$f(\mathbf{x}) = \lambda_0 + \sum_{k=1}^{n} \lambda_k (A_k\mathbf{x}+\mathbf{b}_k), \text{ with } \lambda_0 \geq 0 \text{ and } \forall k, \lambda_k \geq 0.$$

Coefficients λ_k are called Farkas multipliers.

The existence of a set of positive Farkas multipliers, for a given constraints polyhedron, guarantees the function is positive on this polyhedron. In our case, $-\Delta_{p'}^{S'\to T'} - 1$ must be positive on $\mathcal{P}_{p'-1}$. This translates into finding a set of *rational* positive solutions in a system of equalities. This problem is not parametric anymore and may be solved by (non-integral) linear programming in polynomial time. If no such solution can be found, there are violated dependences that we characterize exactly thanks to Figure 6. On the other hand, if a solution is found, there are no violated dependences at depth p' for $\delta_p^{S\to T}$. The problem must then be characterized at depth $p'+1$ where $\mathcal{P}_{p'} = \mathcal{P}_{p'-1} \cap \left\{\Delta_{p'}^{S'\to T'} = 0\right\}$. This does not translate into a costly polyhedral intersection: it is sufficient to add the pair of inequality constraints corresponding to $-\Delta_{p'}^{S'\to T'} = 0$ to the Farkas system at depth $p'+1$. Notice this test is associated with a rational relaxation of the integral constraints on $\mathcal{D}$. This may, in rare cases, lead to a conservative result.[2]

Consider the following example:

[1] To improve readability, we drew a column of zeroes in the original dependence polyhedron to witness the dimension expansion associated with the strip-mine transformation. However, some of the rows have non-zero elements, for instance the part corresponding to A^S. In the actual implementation, the column is, of course, not filled with zeroes.

[2] To guarantee an exact result in all cases, the integer hull of the constraints must be computed beforehand, but this is a combinatorial task.

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

Reversing this loop corresponds to setting $A^{S'} = (-1)$. It is obviously illegal. Let us verify this through our fast legality check. The dependence polyhedron is

$$\delta = \left\{ i \geq 0, i \leq N, i' \geq 1, i' \leq i+1, i' \geq i+1 \right\}.$$

After reversal $-\Delta_1^{S'\to T'} - 1 \geq 0$ is written $i - i' - 1 \geq 0$. Through the Farkas lemma, a necessary and sufficient condition is

$$\begin{aligned} i - i' - 1 \;=\; & \lambda_0 + \lambda_1 i + \lambda_2(N-i) + \lambda_3(i'-1) \\ & + \lambda_4(i'-i-1) + \lambda_5(i-i'+1). \end{aligned}$$

Identifying the coefficients of the i, i', N and constant terms yields the following system:

$$\left\{ \begin{array}{rcl} -1 & = & \lambda_0 - \lambda_3 - \lambda_4 + \lambda_5 \\ 1 & = & \lambda_1 - \lambda_2 - \lambda_4 + \lambda_5 \\ -1 & = & \lambda_3 + \lambda_4 - \lambda_5 \\ 0 & = & \lambda_2 \end{array} \right.$$

The reader may check that this system has no non-negative solution.

5. SCALABILITY

Our dependence analysis is implemented within the modern infrastructure of Open64 and PathScale EKOPath [8]. This compiler family provides key interprocedural analyses and pre-optimization phases such as inlining, interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, as well as induction variable substitution. Our tool extracts large and representative SCoPs for SPEC fp benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop. See [17] for detailed static and dynamic SCoP coverage.

In this section, we exercise this implementation on 6 full SPEC CPU2000 fp benchmarks. These codes were selected because of their large SCoPs, within the set of 8 benchmarks that our tool could handle without instabilities (largely due to the underlying Open64 platform). In the most challenging examples, the biggest SCoP almost contains the whole program after inlining.

Figure 9 summarizes our experimental results. To maximally stress the analyzer, we performed aggressive inlining to favor the formation of the largest possible SCoPs. These statistics are often associated with aggregated SCoPs from multiple functions whose names and line numbers are listed in the second and third columns. #Params gives the number of global parameters, and #Refs gives the total number of array references (read and write) in the SCoP. The next two blocks of columns summarize the properties of the dependence graph, first considering all dependences, second after elimination of transitively covered dependences: #Matrices gives the number of dependence matrices, #Columns give the average number of columns in all dependence matrices (i.e., the average dimension of dependence polyhedra, a good indication of the complexity for Presburger arithmetic), and Analysis Time corresponds the the computation time in seconds to compute the dependence graph on a 2.4GHz Pentium 4 (Northwood) workstation.

The selected SCoPs account for the majority of the execution time in all benchmarks. The smaller SCoPs have been omitted to focus the experiments on the most time-consuming ones. The SCoPs in 168.wupwise feature non-affine array accesses due to

$$
\left(\begin{array}{c|c|c}
\left(\delta_p^{S\to T}\right)_{\bullet,1} \;\Big|\; 0 \;\Big|\; \left(\delta_p^{S\to T}\right)_{\bullet,2..d^S} & \left(\delta_p^{S\to T}\right)_{\bullet,d^S+1..d^S+d^T} & \left(\delta_p^{S\to T}\right)_{\bullet,d^S+d^T+1..d^S+d^T+d_{\mathbf{g}}+1} \\
\hline
\mathrm{SM}_{\mathbf{i}}^S & 0 & \mathrm{SM}_{\mathbf{g}}^S \geq 0 \\
\hline
0 & C_{ond} \geq 0 & 0 \\
\hline
-A^{S'}_{\bullet,1..p'-1} & A^{T_1'}_{\bullet,1..p'-1} & -\Gamma^{S'}_{\bullet,1..p'-1} + \Gamma^{T_1'}_{\bullet,1..p'-1} = 0 \\
\hline
A^{S'}_{\bullet,p'} & -A^{T_1'}_{\bullet,p'} & \Gamma^{S'}_{\bullet,p'} - \Gamma^{T_1'}_{\bullet,p'} \geq 1
\end{array}\right)
$$

Figure 8: Violated dependence at depth $p' > 0$ between S' and T_1'

conservative induction variable substitutions (the actual references are affine but Open64 could not figure it): covered dependences could not be removed in this case.

The full instancewise dependence analysis takes up to 37.512 seconds, for the largest SCoP in 173.applu. This is an extreme case with huge iteration spaces (more than 13 dimensions on average, and up to 19 dimensions). This may sound quite costly, but it still shows that the analysis is compatible with the typical execution time of aggressive optimizers (typically more than ten seconds for Open64 with interprocedural optimization and aggressive inlining and loop-nest optimizations). In all other cases, it takes **less than 5 seconds**, despite thousands of operations on $\mathbb{Z}$-polyhedra with close to 10 dimensions on average.

These results are quite compelling since we compute very large dependence graphs, taking *all pairs of references* into account, with no k-limiting heuristic on their syntactic or nesting distance as it is the case in classical optimization frameworks. Many implementation details can also be improved, using cheap dependence tests as filters for full polyhedral operations, performing on-demand computations on part of the dependence graph only, and improving the polyhedral computation cache to catch a wider scope of operations. These improvements can bring an additional order of magniture acceleration, as shown in previous experiments [42].

According to these results, removing covered dependences is slighlty more expensive. This should not be taken for a definite result: to simplify the implementation, we used polyhedral differences and calls to the PolyLib (following an algorithm closer to the one proposed by Pugh [32]), yet it is well known that an implementation base on Feautrier's PIP would have a much lower complexity [15, 19]. Notice removing covered dependences may sometimes *increase* the total number of matrices, due to domain decompositions to represent non-convex iteration spaces.

A direct computation of the violated dependence graph takes approximately the same amount of time as computing the dependence graph itself. When verifying very complex transformation sequences, it may at most become twice as expensive: this is the case when optimizing the 171.swim benchmark as described in our previous work [9] (leading to 38% speed-up with respect to the peak SPEC performance with the best optimization flags on Athlon64). As described in Section 4.3, if violations are only associated with a limited number of dependences, it is much more practical to apply the fast Farkas-based dependence test and compute violated dependence polyhedra only when a possible violation is detected. This fast dependence test takes a negligible amount of time compared to the actual operations on polyhedra since it considers the same non-negativity constraints but solves (relaxed) rational linear programming problems instead of integral ones.

Finally, we only considered analyses and transformations confined within a given SCoP. These results clearly advocate for extensions of the polyhedral model to irregular programs with complex control structures (e.g., `while` loops). The reader interested in techniques to extend SCoP coverage (by preliminary transformations) or in partial solutions on how to remove this scoping limitation (procedure abstractions, irregular control structures, etc.) should refer to [42, 5].

6. REASONING ABOUT VIOLATIONS

This section builds on graphs of violated dependences to explore transformation correction schemes. Starting from an incorrect transformation sequence, the goal is to reestablish the legality of the final program while disrupting the schedule as little as possible.

The first automatic correction scheme based on instancewise dependence information was proposed by Bastoul and Feautrier [7]. In the general case, they show how to adjust a *partially specified* transformation to respect legality. In the specific case of data locality improvement transformations, they expose liberty degrees which allow them to deeply modify a transformation for legality while preserving the core locality benefits. Yet their method suffers from embarrassingly large constraint systems [7] which may not scale to the size of full SPEC benchmarks. In addition, our correction problem is more general: we are always given a *fully specified*, but illegal transformation, and ought to compute a *minimally intrusive adjustment* to the schedule matrices. Assuming the given transformation is an upper bound to the peak performance achievable for this application, the adjustment to make it correct becomes an optimization problem in itself. This problem is NP-complete in general (e.g., when considering loop fusion/fission as a means to correct the schedule [11]). We are thus working on correction strategies for which a minimal adjustment can be derived effectively, and on sub-optimal heuristics for other correction strategies.

To make the problem and its motivation more concrete, we consider a short example adapted from the actual optimization and correction strategies applied on the 171.swim benchmark. The example in Figure 10 features two loops separated by an intermediate assignment statement, with poor temporal locality on array `A`. Its affine schedule is fully captured by $\beta^{S_1} = (0,0)^t$, $\beta^{S_2} = (1)$, $\beta^{S_3} = (2,0)^t$, every A matrix being the identity matrix of the appropriate dimension and every Γ matrix being 0.

An obvious scheme to improve its locality is to fuse those loops. In our framework, this amounts to setting $\beta^{S_2'}$ to vector $(0,1)^t$ and $\beta^{S_3'}$ to $(0,2)^t$, leaving all other schedule matrices and vectors unchanged. To make this transformation more concrete, we show the result of a naive (unoptimized) code generation in Figure 11.

The transformation is clearly illegal since the dependence from $\langle S_1, N-1\rangle$ to $\langle S_2\rangle$ has been reversed, as well as every dependence from $\langle S_1, i\rangle$ to $\langle S_3, i+1\rangle$. To fix these dependences, it is sufficient to shift the schedule of S_2' by $N-1$ iterations and to shift the schedule of S_3' by 1 iteration. However, naively shifting S_2' makes it necessary to also shift S_3' by $N-1$ (yielding a loop fission and loss of locality). The preferred alternative is to remark that only the iteration $i=0$ of S_3' (after shifting by 1) is concerned by the violated

	Function	Source Lines	#Params	#Refs	All Dependences #Matrices	All Dependences #Columns	All Dependences Analysis / Time (s)	w/o Covered Dependences #Matrices	w/o Covered Dependences #Columns	w/o Covered Dependences Analysis / Time (s)
168.wupwise	zaxpy	11–32	5	16	62	7.5	0.008	62	7.5	0.008
	zcopy	11–24	5	12	30	7.0	0.005	30	7.0	0.005
171.swim	main + calc1 + calc2 + calc3	114–119 + 261–269 + 315–325 + 397–405	5	216	813	10.5	0.895	624	10.4	2.630
172.mgrid	psinv + resid + interp	149–166 + 189–206 + 270–314	2	191	870	8.4	0.735	962	8.4	1.894
173.applu	blts + buts + jacld + jacu + rhs	553–624 + 659–735 + 1669–2013 + 2088–2336 + 2610–3068								
1st SCoP			4	562	3507	11.3	4.420	3188	11.2	14.865
2nd SCoP			2	1983	12814	13.5	37.512	10418	13.5	115.439
200.sixtrack	thin6d	560–588	7	86	158	11.1	0.044	110	11.1	0.117
301.apsi	dcdtz + dtdtz + dudtz + dvdtz + wcont + smth	1326–1354 1476–1499 1637–1688 1779–1833 1878–1889 3443–3448								
1st SCoP			1	275	4264	2.0	0.211	203	2.0	1.750
2nd SCoP			7	198	216	12.9	0.133	207	12.8	0.726

Figure 9: Scalability of instancewise dependence analysis

```
    for (i=0; i<N; i++)
S1    A[i] = ...;
S2  A[0] = A[N-1] ...;
    for (i=1; i<N; i++)
S3    B[i] = A[i-1] ...;
```

Figure 10: Unoptimized code

```
    for (i=0; i<N; i++) {
S'1   A[i] = ...;
      if (i==0)
S'2     A[0] = A[N-1] ...;
      if (i>=1)
S'3     B[i] = A[i-1] ...;
    }
```

Figure 11: Illegal schedule

```
      for (i=0; i<N; i++) {
S''1    A[i] = ...;
        if (i==N-1)
S''2      A[0] = A[N-1] ...;
        if (0<i<N-1)
S''3_1    B[i+1] = A[i] ...;
        if (i==N-1)
S''3_2    B[1] = A[0] ...;
      }
```

Figure 12: Correction

```
S''1   A[0] = ...;
       for (i=1; i<N-1; i++) {
S''1     A[i] = ...;
S''3_1   B[i+1] = A[i] ...;
       }
S''1   A[N-1] = ...;
S''2   A[0] = A[N-1] ...;
S''3_2 B[1] = A[0] ...;
```

Figure 13: Optimized code

dependence from S'_2 and to perform a peeling of S'_3 giving rise to S''_{3_1} and S''_{3_2}. In turn, S''_{3_1} is not concerned by the violation from S''_2 while S''_{3_2} must still be shifted by $N-1$ and eventually pops out of the loop. In the resulting code, the locality benefits of the fusion are preserved and the legality is ensured. To make this adjustment more concrete, the naive code generation is shown in Figure 12. The actual optimized program is shown in Figure 13. Notice that S''_1 is split into 3 parts by the code generation algorithm but this corresponds to a single statement with a single schedule in our representation. Only S''_{3_1} and S''_{3_2} represent resulting statements with different schedules.

7. CONCLUSION

Instancewise array dependence analysis computes a finite representation of the set of all pairs of dependent iterations of all statements. This problem has always been considered non-scalable or an overkill compared to less expressive but faster dependence tests. We presented technical contributions to instancewise array dependence analysis, and compelling experimental evidence of its scalability through the first validation on full SPEC CPU2000 benchmarks. In addition, we demonstrated how our approach allows to reason about violated dependences across arbitrary transformation sequences. This relieves the compiler of the expensive and cumbersome task of implementing specific legality checks for each single transformation. It also allows, in the case of invalid transformations, to precisely determine the violated dependences that need to be corrected. Identifying these violations can in turn enable automatic correction schemes to fix an illegal transformation sequence with minimal changes. This automatic correction approach is a promising way to design future optimization frameworks, allowing profitability heuristics to operate on simplified search spaces, decoupling precise legality enforcement from the most combinatorial optimization process.

8. REFERENCES

[1] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, october 1987.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[3] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[4] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[5] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles, France, Feb. 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, Sept. 2004.

[7] C. Bastoul and P. Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005.

[8] F. Chow. Maximizing application performance through

interprocedural optimization with the pathscale eko compiler suite. `http://www.pathscale.com/whitepapers.html`, Aug. 2004.

[9] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Intl. Conf. on Supercomputing (ICS'05)*, pages 151–160, Boston, Massachusetts, June 2005.

[10] J.-F. Collard. *Parallélisation automatique des programmes à contrôle dynamique*. PhD thesis, Université Pierre et Marie Curie (Paris 6), France, Jan. 1995. `http://www.prism.uvsq.fr/~jfc/memoire.ps`.

[11] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[12] C. Eisenbeis and J.-C. Sogno. A general algorithm for data dependence analysis. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 292–302, Washington, D. C., United States, 1992. ACM Press.

[13] P. Feautrier. Array expansion. In *ACM Intl. Conf. on Supercomputing*, pages 429–441, St. Malo, France, July 1988.

[14] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

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

[16] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[17] 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. *Intl. J. of Parallel Programming*, 2006. 57 pages. Accepted for publication.

[18] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, New York, june 1991.

[19] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[20] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[21] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[22] X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 International Conference on Parallel Processing*, pages 204–211, St. Charles, august 1990.

[23] L. Lamport. The parallel execution of do loops. *Communications of ACM*, 17(2):83–93, 1974.

[24] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3):649–671, 1998.

[25] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[26] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Intl. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.

[27] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20thACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[28] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1991.

[29] P. Petersen and D. Padua. Experimental evaluation of some data dependence tests. Technical Report CSRD 1080, University of Illinois at Urbana-Champaign, february 1991.

[30] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1121–1132, november 1996.

[31] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3):196–213, march 2004.

[32] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, Aug. 1991.

[33] W. Pugh. Uniform techniques for loop optimization. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, pages 341–352, Cologne, Germany, June 1991.

[34] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5):773–815, Sept. 2000.

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

[36] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, UK, 1986.

[37] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, Mar. 2006. Springer-Verlag.

[38] F. Vivien. *Détection de parallélisme dans les boucles imbriquées*. PhD thesis, ENS - Lyon, 1997.

[39] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1995.

[40] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[41] M. Wolfe and C. W. Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, 1992.

[42] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.