Simplification and Run-time Resolution of Data Dependence Constraints for Loop Transformations
Diogo Sampaio, Louis-Noël Pouchet, Fabrice Rastello

To cite this version:

HAL Id: hal-01653819
https://inria.hal.science/hal-01653819
Submitted on 5 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations

Diogo N. Sampaio
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble, France
diogo.nunes-sampaio@inria.fr

Louis-Noël Pouchet
Colorado State University
Fort Collins, USA
pouchet@cse.ohio-state.edu

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble, France
fabrice.rastello@inria.fr

ABSTRACT
Loop transformations such as tiling, parallelization or vectorization are essential tools in the quest for high-performance program execution. Precise data dependence analysis is required to determine whether the compiler can apply a transformation or not. In particular, current static analyses typically fail to provide precise enough dependence information when the code contains indirect memory accesses or polynomial subscript functions to index arrays. This leads to considering superfluous may-dependences between instructions that prevent many loop transformations to be applied.

In this work we present a new hybrid (static/dynamic) framework that allows to overcome several limitations of purely static dependence analyses: For a given loop transformation, we statically generate a test to be evaluated at runtime. This test allows to determine whether the transformation is valid, and if so triggers the execution of the transformed code, falling back to the original code otherwise. Such test, originally constructed as a loop-based code with $O(n^2)$ iterations (n being the number of iterations of the original loop-nest), are reduced to a loop-free test of $O(1)$ complexity thanks to a new quantifier elimination scheme that we introduce in this paper. The precision and low overhead of our method is demonstrated over 25 kernels.

CCS CONCEPTS
• Computing methodologies → Symbolic and algebraic algorithms; • Software and its engineering → Compilers;

KEYWORDS
Hybrid Dependence Analysis, Loop optimization, May-alias, Polynomials, Polyhedral Transformation, Quantifier Elimination

ACKNOWLEDGMENTS
This work was funded in part by the U.S. National Science Foundation, award 1524127 and by CAPES - Brazil. The authors are also very grateful to Alain Ketterlin, who developed ASTER.

1 INTRODUCTION
Loop optimizations span from vectorization, scalar promotion, loop invariant code motion, or software pipelining to loop fusion, skewing, tiling [59, Ch.9], or loop parallelization. The polyhedral compilation framework [6, 18] subsumes most of these loop transformations in a single, unified formalism that abstracts in an algebraic way the iteration space of multidimensional loops and the dependences between instructions. Polyhedral program transformation has reached a sufficient level of maturity and software robustness to now be integrated in production compilers, as exemplified with LLVM (Polly [23]), GCC (Graphite [37]), or IBM XL/C [29]. But to date, these optimizers are effective only in restricted cases and are severely challenged by polynomial array subscript functions and may-aliasing issues, for instance.

Generally speaking, integrating advanced loop transformation systems in production compilers involves solving problems inherent to the compiler intermediate representation (IR) lowering: array linearization, type erasure and pointers with possibly multiple indirections makes the task of optimizing loops hard. The main reason is that most loop transformations require precise data dependence analysis that is expressed between loop iterations. When employing generic IRs (e.g., LLVM IR [40] or GIMPLE [21]), semantic key information, such as multidimensional array references, are removed (if not already absent at source level), significantly decreases the effectiveness of static analyses.

Static dataflow analysis [13, 17] based on polyhedral abstraction [6, 18] succeeds in computing precise dependencies for array subscript references that use affine access functions of the loop indices [23, 46]. But, whenever their is no known relationship between the base-pointers, or whenever the access functions are not affine, existing loop-based data dependence analyses fail to compute exact dependences. In those very common scenarios, the compiler will conservatively choose not to implement any loop transformation. This is disheartening given the numerous demonstrations of how polyhedral optimizers can produce high-performance code [6, 10, 22, 32, 36, 39, 52] by exposing parallelism and data locality.

Vectorization is an example of loop optimization that suffers from poor alias information: as most current architectures support SIMD instructions (e.g. AVX), enabling aggressive loop vectorization [35] in production compilers became a critical concern. In many cases
production compilers where not able to implement SIMD vectorization. The reason was the presence of spurious dependencies due to conservative assumptions by the compiler about potentially overlapping memory regions being referenced (i.e., aliasing). The practical observation of the inefficiency of static alias analysis led to the development of a specific disambiguation technique for vectorization [34, 43], to enable efficient SIMDization. As opposed to speculative pointer disambiguation, this approach uses code cloning and versioning and determines at runtime which code version can be safely executed.

In a similar spirit, we develop a may-dependence violation checking technique that allows to safely apply loop transformations even in the presence of may-aliasing arrays with polynomial accesses functions. This combination of static analyses and runtime checks allows to assert the correctness of the transformation with regards to (runtime) data dependencies, falling back to the non-transformed code if the runtime validity test fails.

Two instructions are (data-)dependent if they both access the same memory cell and at least one of them is a write. To preserve semantics, a transformation must preserve the relative execution order of dependent instructions: the producer shall be executed prior to the consumer. One can create a simple runtime check to determine whether all dependencies are preserved after transformation, as illustrated below. Let us consider a simple abstract example to illustrate this point (where \(S(i,j)\) is a statement at iteration \(i,j\) that writes to memory address \(f(i,j)\), where \(f\) is unknown at compile-time):

\[
\begin{align*}
\text{for } (i = 0; \ i < n; \ i++) & \\
\text{for } (j = 0; \ j < n; \ j++) & \\
S(i,j); & \\
\end{align*}
\]

Suppose we want to permute the two loops:

\[
\begin{align*}
\text{for } (j = 0; \ j < n; \ j++) & \\
\text{for } (i = 0; \ i < n; \ i++) & \\
S(i,j); & \\
\end{align*}
\]

This loop interchange can be modeled in the polyhedral framework by applying a new schedule \(T(i,j) = (j, i)\) on the input program (see Sec. 4). To test the validity of \(T\) at runtime, one can establish the non-existence of a dependence violation by testing, for each pair of loop iterations, whether (a) they access the same memory cell, and (b) if so whether they are executed in the same order as in the original program. This would lead to the test:

\[
\begin{align*}
\text{valid} &= \text{true} \\
\text{for } (i = 0; \ i < n; \ i++) & \\
\text{for } (j = 0; \ j < n; \ j++) & \\
\text{S(i,j);} & \\
\text{valid} &= \text{false} \\
\end{align*}
\]

Where \(<_{lex}\) is a lexicographic comparison, as detailed in Sec. 3.

This approach is prohibitively expensive, as it amounts to inspect at runtime a number of iterations that is quadratic with the number of iterations of the original loop nest. The purpose of our approach is to implement similar checking capability, but reduce to a \(O(1)\) runtime check, over loop-invariant variables. This leads to a fundamental challenge: the need to be able to perform quantifier elimination (e.g., eliminating loops in the above test), while operating on integer multivariate-polynomials and without losing accuracy through loose relaxation techniques. In this paper we present an end-to-end solution to these challenges. Our contributions are summarized as follows.

- A new quantifier elimination method for integer-valued polynomial constraints.
- A method to generate simple runtime tests for a variety of critical loop transformations, such as tiling, parallelization, vectorization, etc. to ensure the legality of these transformations.
- An extensive evaluation demonstrating that our elimination scheme is precise and our approach has negligible runtime overhead, thus enabling optimizations.

Sec. 2 shows practical examples motivating the need for our approach, and Sec. 3 provides the theoretical foundations on the program representation we use. Sec. 4 presents the expression of runtime checks for numerous key program transformations, while Sec. 5 presents the variable elimination techniques for polynomial constraints. Experimental results are presented in Sec. 6, and related work is discussed in Sec. 7 before concluding.

2 MAY-DEPEND ILLUSTRATING CASES

In this section we go through a few motivating examples for which state-of-the-art static analyses are typically not able to disambiguate some dependences, and thus prevent compilers from implementing aggressive program optimizations. Our proposed runtime dependence test, presented in Sec. 4 and beyond, can handle all these cases even for complex loop transformations.

In-place block xMADD. We consider a multi-dimensional data space (e.g. a 2D-matrix, an image, etc.) and a simple kernel that performs a block-computation (e.g., addition of two sub-matrices) that can possibly be in-place Computations restricted to a sub-block (or “window”) of a multi-dimensional array can be found in numerous function libraries. Linearized arrays (see below) can also occur in such codes (e.g. OpenCV [31]).

\[
\begin{align*}
\text{void xMADD} & \quad (\text{int } M, \text{ int } N, \text{ int } K, \text{ double } A[M][N], \text{ int ai, int aj, double } B[M][N], \text{ int bi, int bj}) \\
& \quad \text{for } (i = 0; \ i < K; \ i++) \\
& \quad \text{for } (j = 0; \ j < K; \ j++) \\
& \quad A[ai+i][aj+j] = (B[bi+i][bj+j]*B[bi+i+1][bj+j] + B[bi+i+1][bj+j+1]) / 4 \\
\end{align*}
\]

In this example, there is no restriction on \(A\) nor \(B\), and actually they could be equal. In this context, there is a may-dependence between all read accesses and all write accesses. Output dependencies between accesses to \(A\) are the only ones that can be disambiguated statically. Indeed, in the case where \(A = B\), depending on the respective values of \(ai, aj, bi,\) and \(bj\), any overlap is possible. Hybrid alias analysis as developed in most production compilers for auto-vectorization [34] could disambiguate the innermost loop as follows 1 (considering vectors of length 4 and \(K\) a multiple of 4):

\[
\begin{align*}
& \text{for } (i = 0; \ i < K; \ i++) \\
& \quad \text{if } (\text{(as}_{ai+i}+ai+i)N+aj < 2) \Rightarrow (B[\text{bi+1}][bi+1]+B[\text{bi+1}][bj+2]) \} \\
& \quad (\text{as}_{ai+i}+ai+i)N+aj < 2) \Rightarrow (B[\text{bi+1}][bi+1]+B[\text{bi+1}][bj+2]) \} \\
& \quad \text{for } (j = 0; \ j < K; \ j++) \\
& \quad \text{... / original version} \\
& \quad \text{else} \\
& \quad \text{for } (j = 0; \ j < K; \ j++) \\
& \quad \text{... / vectorized version}
\end{align*}
\]

1Note that accesses are shown linearized here as they would appear in the compiler intermediate representation.
The idea of this technique is to check the overlapping of accessed areas. As an example the first equality expresses the overlapping of intervals A[i+a][j+j+ 0 .. 3] and B[i+b][j+j+ 0 .. 3]. Apart from the potential complexity of the runtime test in the cited technique (due to the absence of algebraic simplifications), no existing technique can disambiguate the block regions necessary for register tiling. The main reason is the multi-dimensionality of data structures.

FW with Triangular Matrix. Here we consider a symmetric matrix (representing for example a tensor with symmetries) stored in a compact way. T[1, j] is represented as T[1*(1-1)/2+j] if i ≥ j and T[j*(j-1)/2+i] otherwise. Such matrix representation (named “packed”) is used by most high-performance linear algebra libraries such as LAPACK [37], or MKL [30] for triangular, symmetrical, or Hermitian forms. Boost [7] also uses packed representation for banded matrices. The example used in later Sec. 3 which corresponds to one part of the compilation of Floyd-Warshall with symmetries uses this representation. It turns out that such code would highly benefit from tiling [60], as the data reuse available has similarities with matrix-matrix multiplication. However, none of the existing dependence analyses (even hybrid ones) can disambiguate this code. The reason is the presence of polynomials in access functions. Obviously delinearization techniques that are based on pattern matching cannot infer the 2D structure of memory accesses [25]. Any compiler will fail at tiling this loop nest.

ADI with Linearized Array. Here we consider some computation on linearized arrays. An access to cell (j, i) of array τ[H][H] is written as τ[H*i+j+1]. Multi-dimensional array linearization is widely present in high-performance legacy code. One of the reason is that languages such as C99 or Java (e.g. Colt [14], EJML [16]) implement multi-dimensional arrays as pointers of pointers (even for fixed-size arrays in Java) which turns out to be less efficient than the equivalent linearized version. Also, even if some programming languages support packed multi-dimensional arrays, some compiler intermediate representations (e.g. LLVM-IR [38]) will not support it and linearization is performed early in the compilation stages. As an example we consider a simple 2D loop nest that interleaves two sweeps, one on each direction.

For the same reason than for the previous example (FW), as linearization introduces polynomials, static dependence analysis will fail in disambiguating the memory accesses. Also, depending on the actual values of H and n at runtime, loop interchange might not be valid. Without code cloning, the compiler has to be conservative regarding the presence of data dependencies and cannot fully optimize the loop nest.

3 THEORETICAL FOUNDATIONS

To illustrate our notations and terminology, consider the example code below. We consider nested loops so that any statement is uniquely identified by the iteration vector ((k, i, j) here).

Without loss of generality, we use a canonical representation of the nested loop [18] where to each iteration vector is associated an unique statement instance. As an example the previous example is equivalently modeled as follows:

In this form any statement is indexed by a vector of possibly larger size than the original loop depth ((i, k, i, j) here). The iteration domain of a nested loop is a geometric representation of all possible values of loop indices. It can be an over-approximation of it [3]. In our example, the iteration domain for statement S1 is D_{S1} = \{(t, k, i, j), t = 0 \land 0 \leq j \leq i < k < n\}. In this paper we restrict the iteration domain to be an union of convex polyhedra, in other words it can be represented as a disjunctive normal form of affine inequalities.

A schedule is expressed as a function (say TS1 for statement S1) from D to D’ \in Z^2 a vector space of dimension s ≥ 1 (Z represents the integers). TS1(0, k, i, j) = TS1(0, k’, i’, j’) means that S_{10,0, k,i,j} and S_{10, k’, i’, j’} are executed in parallel. TS1(0, k, i, j) <_{lex} TS1(0, k’, i’, j’) means that S_{10, k, i, j} is executed before S_{10, k’, i’, j’}. A loop transformation is expressed as a schedule, and most composition of loop transformations can be represented as affine schedules [6, 18, 22]. For example, interchanging loops i and j for the loop nest encapsulating S1 would lead to TS2(0, k, i, j) = (0, k, j, i). The canonical form allows to express the schedule of the original code as the identity function from D to D.

A data dependence between two statement instances expresses an ordering (represented as ≺_{d}) constraint of their respective execution. A data dependence is due to read and write accesses to the same memory location. Here S_{10, k, i, j} writes to location W_{i}(0, k, i, j) = T + (i-i-1)/2 + j and reads to locations R_{2}(0, k, i, j) = T + (i-i-1)/2 + j, R_{3}(0, k, i, j) = T + k(k-1)/2 + i, and R_{4}(0, k, i, j) = T + k(k-1)/2 + j. There is a read-after-write (flow) dependence between S_{10, k, i, j} and S_{10, k’, i’, j’} from W_{1} to R_{2} as soon as: (1) (0, k, i, j) \in D strictly precedes (0, k’, i’, j’) \in D in the original schedule; which is written as (0, k, i, j) ≺_{lex} (0, k’, i’, j’) \Rightarrow (0, k’, i’, j’) \in D (\text{lexographic comparison}); (2) read/write locations overlap; which is written (for atomic data) as W_{1}(0, k, i, j) = R_{2}(0, k’, i’, j’). As an example, S_{10, 3, 2, 1} ≺_{d} S_{10, 4, 2, 1} as W_{1}(0, 3, 2, 1) = R_{2}(0, 4, 2, 1) = T + 4. A read-after-write is a flow dependence; A write-after-read is an output dependence; A write-after-read is an anti-dependence.

\[
\begin{align*}
&\text{for } (k = 3; k < n; k++) \quad // j < c_k \\
&\text{for } (i = 1; i < k; i++) \\
&\text{for } (j = 0; j < k; j++) \\
&S_1: d[i*(i-1)/2+j] = d[k*(k-1)/2+i] \land d[k*(k-1)/2+j]\; // j < c_k \\
&\text{for } (i = k+1; i < n; i++) \\
&\text{for } (j = 0; j < k; j++) \\
&S_2: d[i*(i-1)/2+j] = d[1*(1-1)/2+i] + d[k*(k-1)/2+j];
\end{align*}
\]
For any schedule to be valid it must preserve all such dependences. As an example, for the previously considered loop interchange $T_S(0, k, i, j) = (0, k, j, i)$, the dependence $S_{10,3,2,1} \leq_d S_{10,4,2,1}$ is preserved as $T_S(0,3,2,1) = (0,3,1,2) \prec_{\text{lex}} T_S(0,4,2,1) = (0,4,1,2)$. The loop interchange is valid as soon as all dependences are preserved.

4 DEPENDENCE VALIDITY TESTS

Our approach is based on checking at runtime whether the optimized or the original code should be executed. The transformation is correct (with regard to data dependencies) as soon as the transformed schedule does not contain any violated dependence. The following system of equations, where a dependence is represented as a point $(v, v')$ in a multi-dimensional space, defines the exact set of violated dependencies. Let us call this set, the violated set. Here, existential quantifiers are used to express non-emptiness of this set. The contribution of this work is to eliminate those quantifiers.

$$\bigvee_{(A, A')} \in mD \exists (v, v') \in \mathbb{Z}^{2d} \text{ s.t.}$$

(exist. quantifier)

$$\forall v \in D_A \land v' \in D_{A'}$$

(domain)

$$\land v \prec_{\text{lex}} v'$$

(orig. sched.)

$$\land A(v) = A'(v')$$

(same access loc.)

$$\land T_A(v) \prec_{\text{lex}} T_{A'}(v')$$

(violation)

where:

1. $d$ is the dimension of the loop nest in canonical form;
2. $A$ and $A'$ are access functions (e.g. $W_i$ and $R_2$ in the example of Section 3), one of them necessarily being a write access (to generate a may-dependence);
3. $D_A$ (resp. $D_{A'}$) is the set of points in the domain where $A$ (resp. $A'$) is accessed (access $D_{S_1}$ for $W_i$ and $R_2$ in our example); $mD$ represents all possible pairs of such functions that could lead to a may-dependence (a dependence the compiler cannot disambiguate statically);
4. $T_A$ (resp. $T_{A'}$) represents the schedule of the domain including access $A$ (resp. $A'$) after transformation ($T_S$ in our example).

Observe that emptiness (overall condition with quantifiers that evaluate to false) of the violated set is a sufficient condition for the transformation to be valid. So it is safe to over-approximate it (it is obviously not to under-approximate it), but if too loose it may lead to conservatively decide not to apply an even safer transformation. In that sense, precision (tightness of the over-approximation) of the quantifier elimination process is important to make the overall approach of hybrid analysis effective. We now illustrate this system of equations through different loop transformations and applications.

Interchange transformation can be used to expose parallelism and locality. Take ADI as an example and the may-dependence $(A, A') = (R_2, W_i)$, where $A(i, j) = R_2(i, j) = -H + Hj + i$ and $A'(i, j) = W_i = Hj + i$. Consider loop interchange as the desired transformation, i.e. $T(i, j) = (j, i)$. The system of equations writes as follow:

$$\exists (i, j, i', j') \in \mathbb{Z}^4 \text{ s.t.}$$

$$1 \leq i < m \land 1 \leq i' < m$$

(domain)

$$\land 1 \leq j < n \land 1 \leq j' < n$$

(domain)

$$\land (i, j) \prec_{\text{lex}} (i', j')$$

(orig. sched.)

$$\land -H + Hj + i = Hj' + i'$$

(same access loc.)

$$\land (i, j) \prec_{\text{lex}} (i', j')$$

(violation)

$(i, j) \prec_{\text{lex}} (i', j')$ writes as $i < i'$ or $(i = i' \land j < j')$ and thus (orig. sched.) $\land$ (violation) simplifies to: $i < i' \land j' < j$.
Loop-Body Scheduling / Software Pipelining. As observed above, the necessary condition for vectorization of size \( I \) to be valid is that all loop-carried dependences must be of distance greater or equal to \( I \). Dependence distance (which actually corresponds to a lower-bound, not to an exact distance) is used by standard loop-body scheduling used for EPIC architectures. For any given dependence distance, the system of inequalities is very similar to the one constructed for vectorization.

Tiling / Permutability. The legality of loop tiling can be assessed by detecting that a set consecutive loops are permutable, and tiling can then be implemented in a general way \([6]\). Tiling exposes locality and parallelism simultaneously at several levels. Rectangular tiling (possibly after some skewing) is a very powerful loop transformation \([60]\). Let us consider again the example given in Section 3 and suppose we want to tile the loop nest enclosing \( S_1 \). The band \((k, i, j)\) of the domain \( D_{S_1} = \{(0, k, i, j) : 0 \leq j < i < k < n\}\) is fully permutable if individually \((k, i, j)\), \((k, i, j)\), and \((i, j)\) can be interchanged \([48]\). The concept can actually be expressed in a more condensed way \([2]\): all dependences should be within the positive cone. Restricting to may-dependence \((\mathcal{W}_1, \mathcal{R}_3)\)
\[
\mathcal{W}_1(0, k, i, j) = \left(i(i-1)/2 + j, \mathcal{R}_3(0, k, i, j) = k(k-1)/2 + i, \right.\]
\[\text{the system of inequalities is written as follows:}\]

\[
\exists (t, k, i, j, t', k', i', j') \in \mathbb{Z}^6 \text{ s.t.}
\]
\[
t = 0 \leq j < i < k < n \quad \text{(domain)}
\]
\[
t' = 0 \leq j' < i' < k' < n \quad \text{(domain)}
\]
\[
\wedge (t, k, i, j) \preceq_{\text{orig.}} (t', k', i', j') \quad \text{(orig. sched.)}
\]
\[
\wedge i(i-1)/2 + j = k(k'-1)/2 + i' \quad \text{(access loc.)}
\]
\[
\wedge (k, i, j) \not\preceq_{\text{permute}} (k', i', j') \quad \text{(violation)}
\]

where \((k, i, j) \preceq_{\text{permute}} (k', i', j')\) writes as \(k \leq k' \land i \leq i' \land j \leq j'\). Note that if we want to combine tiling with another transformation \(T\) (e.g. skewing prior to tiling) a simple way is to check validity of \(T\) and then check the permutability of \(T\). The (violation) constraint would rewrite as:

\[
T(0, k, i, j) \not\preceq_{\text{permute}} T(0, k', i', j') \vee T(0, k, i, j) \not\preceq T(0, k', i', j')
\]

Finally, we remark that any loop transformation sequence that can be expressed by an affine multidimensional schedule (e.g., \(T\)) can be verified this way. That is, our approach can handle a wide variety of complex compositions of loop transformations, including interchange, skewing, fusion, distribution, shifting/re-timing, peeling, vectorization, parallelization, etc.

5 QUANTIFIER ELIMINATION

This section describes our elimination scheme that allows to reduce the complexity of the validity test to \(O(1)\). The presence of an existential quantifier means that a loop is needed at runtime to explore possible values. Eliminating a quantifier eliminates the need for the corresponding loop. Geometrically speaking, the quantifier elimination corresponds to perform an orthogonal projection of the violated set along the eliminated dimensions. As already mentioned this set can be safely over-approximated and so is its projection. But the tightest the better, and as we will see further, real-valued relaxation (our initial problem is an integer-valued) usually leads to too loose test.

Our method to perform quantifier elimination is an extension of the Fourier-Motzkin Elimination (FME) method that handles integer multivariate-polynomials. Figure 1 presents an overview of the process. For this section, let \(S(P, V) : \{E_0, E_1, \ldots, E_n\} \geq 0 \land \left\{E'_0, E'_1, \ldots, E'_m\right\} = 0\) be our system of constraints. \(E\) and \(E'\) are expressions over \((P, V)\). \(V\) represents the set of quantifiers (loop variants / variables) to be eliminated. \(P\) a set of free-variables (loop invariant / constants). Our quantifier elimination will project \(S(P, V)\) into \(S'(P)\).

Schweighofer Tester. One of the most important part of the algorithm relies in the ability to prove, whenever possible, the sign of an expression in a given system of constraints. To this end we use the theorem described by \([54]\) to evaluate expressions’ signs under a system of constraints \(3\).

Given a system of inequalities \(S : \{E_1, E_2, \ldots, E_n\} \geq 0\), Schweighofer provides a method to evaluate if the inequality \(I : E \geq 0\) is implied by \(S\). If \(I\) can be written as the sum of the product of any power of the inequalities of \(S \cup \{1\}\) multiplied by non-negative factors (multivariate-polynomial in \(E_1, \ldots, E_n\) with positive coefficients), then \(S\) implies \(I\). As an example, if the system \(S = \{E_1 = -xy - x \geq 0 \land E_2 = x - 1 \geq 0 \land E_3 = y \geq 0\}\) is true, then \(y \geq 1\). Indeed, \(y = 1 - 2E_3 + E_2 + E_1 + E_3\). Unfortunately, raising a subset \(S' \subseteq S\) of cardinality \(m\) to degree \(d\), leads to a polynomial (in \(E_1\)) with a number of terms combinatorial with \(m\) and \(d\). Take as an example the subset \(S' = \{E_1, E_2\}\) (we have \(m = 2\)) and \(d = 2\).

The goal (to prove the positivity of \(E\)) is to find coefficients \(\lambda_i\) of the polynomial \(\lambda_0 + \lambda_1E_1 + \lambda_2E_2 + \lambda_3E_1E_2 - E = 0\) that are positive. This is done by expanding the polynomial and building a linear program by matching the coefficient of each of its term to 0. In our case, the expanded form \((\lambda_0 + 1) + (-\lambda_1 + \lambda_2 + \lambda_12)x - y = 0\) leads to the linear program \(\lambda_0 = 0 \land \lambda_0 + 1 \geq 0 \land -\lambda_1 + \lambda_2 + \lambda_12 = 0 \land -1 = 0 \land \ldots\) that has no solution in that case. Our implementation chooses, heuristically, subsets \(S' \subseteq S\) of bounded cardinality \(m\) and limits the maximum degree \(d\) to which \(S'\) are raised. Solving the LP is heuristically done by using a linear solver where positiveness of the \(\lambda_i\)’s is enforced using a simple heuristic. Whether the precision gain using a LP solver is worth compared to the overhead has to be determined. Observe that the result to the LP possibly gives a more accurate inequality as it allows (in case of success) to prove that \(E \geq \lambda_0\). As discussed further maximizing \(\lambda_0\) can be used as the objective function of the linear program when looking for tight bounds.

The sign detection of an expression \(E\) is done by evaluating if \(S\) implies \(E \geq 0\) or \(S\) implies \(-E \geq 0\). When both are implied, if one of the inequalities can be made strict, then \(S \equiv false\), otherwise \(E = 0\). Whenever none of the inequalities can be proven to be implied by any \(S'\), \(E\) is factorized, if possible, and its sub-expressions are recursively evaluated, with particular rules for exponential, multiplication and sum. To avoid replicated sign tests a cache of signs per expression is kept for each Schweighofer Tester. Observe that each system of inequalities holds its own tester, and whenever the system changes, the tester must be rebuilt, clearing the cache.

Fourier-Motzkin Lowering (FML). The Fourier-Motzkin elimination (FME) is a well known algorithm for removing variables of linear inequality systems on \(\mathbb{R}\). For a given linear system \(S\), the

\[\text{We actually use the technique in a more general context where we do not enforce variables to live in a compact polytope.}\]
For each system $S_i$ and $S_j$: Select variable with lowest $t$, lowering its degree as target $t$

Check Signs
For all inequalities $F.t \geq P$ Decide sign of $F$

All $F$ known and $=0$:
unknown $F$ or it might be $0$

Equality Factorize
Exists $F.t = P$ with $F > P^t$ Then add $t = 0 \land P = 0$

Equality Substitute
Exists $t = E$? Replace $t$ by $E$

Normalize
Exists $F.G \geq H$ with $F > H$? Replace it by $F > H \land E > 0$

Simplify
Exists $E$ such that $S_i \land E$ implies $E$? Remove $E$

Prune $S_i$ implies $S_2$? Remove $S_1$

Output: $S_1'$ or $S_2'$ over Parameters

where each $L_j$ and $U_j$ have degree in $t$ (say $t$-degree) strictly lower than $\alpha$. In both modes, inequalities with non-null coefficients $C_i$ are normalized to the form

$$E(P, V) \leq U_i'(P, V)$$

or $L_i'(P, V) \leq E(P, V)$

where $E(P, V)$ corresponds to a common multiple (ideally lowest) of the (non-null) middle parts. Note that only Convergence mode leads to lower the degree for sure. Initial steps start in Precision mode and switches to Convergence whenever it does not converge to eliminating $t$.

Writing inequalities in this form requires to compute the sign of the coefficient expressions $C_i$. Failing do determine a strict sign for a given $C_k$ leads to split the system in two (when sign is known but is not strict) or three (when the sign is unknown) new systems, each holding a distinct constraint on $C_k$:

$$S_{>0} : S \land (C_k > 0)$$
$$S_{=0} : S \land (C_k = 0)$$
$$S_{<0} : S \land (C_k < 0)$$

In theory the number of generated systems by doing this split is exponential with the number of unknown coefficients retrieved. It turns out that the sign can usually be proved, and no exponential growth is observed in practice. If all coefficients signs are determined and strictly non-zero, then the system does apply the lowering step (Eliminate).

Call $S'$ the new system obtained from $S$ after a lowering step. Obviously $S'$ contains all the combined constraints $L_i' \leq U_i'$. But there is a subtlety concerning the constraints from $S$ reported in $S'$.

In Precision mode, all constraints from $S$ that do not increase the $t$-degree of $S'$ are reported in $S'$. In Convergence mode, all constraints from $S$ with $t$-degree smaller than $\alpha$ are reported in $S'$.

Pre-conditioning. Prior to performing FML, a conjunctive system undergoes a few transformations:

(1) Convex Hull: The system is enriched with an octagon convex hull. For each couple of variables $u \neq v$, if a constant $c$ is found (only the tightest is kept) such that $u - v \geq c$ (or, $u + v \geq c$, or $v \leq c$, or $-v \leq c$) can be proven to be implied by $S$, then this inequality is added to the system. This aids the Schweghofer Tester sign detection process to overcome the limits imposed on $m$ and $d$.

(2) Equality Factorize: Whenever an equality can be written as $f(P) \cdot g(P, V) = h(P, V)$ and $|f(P)| > |h(P, V)|$ can be proven, then (as we are in $\mathbb{Z}$) the equalities $h(P, V) = g(P, V) = 0$ are added to the system.

(3) Equality Substitute: For equalities such that a variable $t$ in $V$ can be isolated in the form $t = E(P, V \setminus \{t\})$, $t$ is eliminated from the system by replacing it by $E(P, V \setminus \{t\})$ in all other constraints of the system (the equality that becomes the identity is removed). Failing to apply this rule, any equality $E(P, V) = 0$ is replaced by two inequalities $E(P, V) \geq 0 \land E(P, V) \leq 0$.

(4) Normalize: Applying FME method on an integer system might suffer from loss of precision. For example, in the case: $i \leq 100t \leq 100(j-1)$. The elimination of $t$ would lead to $i \leq 100(j-1)$. In an integer system, normalization [51] is an important step to improve precision. In this example, $100t \leq 100(j-1)$ can be normalized to $t \leq j + [-1/100] = j - 1$, which leads to $i \leq 100(j - 1)$ after the

Figure 1: Quantifier Elimination scheme. Input and output are sets of conjunctive systems of polynomial inequalities. “A” stands for affine hyper-planes.
elimination. Our normalization extends the one from [51] to non-numerical coefficients. For any inequality in the form \( P(P) - G(P, V) \geq H(P) \), if Schweighofer Tester proves that \(|P(P)| \geq |H(P)|\), then the inequality becomes \( \pm G(P, V) \geq |\pm 1, 0| \), depending on the signs of \( F \) and \( H \).

After applying Pre-conditioning, if the system has no more free-variable, the process branches to Simplify phase. If some free-variables remain, but changes where made to the system, the entire process is restarted (Schweighofer Tester). If no changes have been performed then it proceeds to the variable elimination stage (Fourier-Motzkin Lowering).

Simplify. Once all quantifiers have been eliminated, a last phase consists in: 1. simplifying each inequality; 2. eliminating redundant ones or proving its contrary. Hence, 1. Function Simplify takes each inequality \( E \geq 0 \) of the system and factorizes it (as much as possible) into \( E = \prod F_i \geq 0 \). Then, any factor \( F_i \) for which the (strict) sign can be proven (using the other inequalities) is removed. Observe that the orientation of the inequality can be potentially reversed depending on the number of negative factors. 2. Once this is done, for every inequality \( I : E \geq 0 \) we test if \( \{ S \setminus F \} \) implies \( I \). If it does then \( I \) is removed; If it can be proved that \( \{ S \setminus F \} \) implies \( E < 0 \), then a contradiction \( (I \land \neg I) \) is found, and the system is turned into false.

Prune. The last “simplification” step acts over the disjunction formed by all systems \( S_0^2(P) \). Using the Schweighofer Tester, if a system \( S_0^2 \) implies all inequalities of a system \( S_1^2 \), then \( S_1^2 \) is removed. All pairs are tested.

The output of this method is a disjunctive normal form of systems over the parameters \( P \). If (only) a single one of these systems evaluates to true, then it is unsafe to use the optimized version of the loop.

6 EXPERIMENTAL RESULTS

Framework implementation. We recall the three steps of our framework. Given a (sequence of) loop transformation(s) that forms the tentatively optimized program, and a representation of the original program as described in Sec. 3, our framework: (1) builds sufficient conditions to prove the validity of the transformation with regard to may-dependences; (2) eliminates loop indices in the constructed validity test; (3) incorporates the simplified test at the entry point of the program region, deciding on branching to the optimized version or fallback to the original (un-optimized) program version.

We have implemented an end-to-end prototype of the framework presented in this paper. Our prototype can handle C code with loops and array access functions containing polynomial expressions. ASTER extracts the program representation. PoCC computes a program transformation for this representation and emits ASTER directives to describe its transformation. ASTER generates the complete validity test (with complexity in the order of \( O(n^2d) \) where \( d \) is the loop depth), which is fed to our implementation of the quantifier elimination strategy. Glue scripts are used to create a binary using versioning, containing the optimized code and the original one (fallback), the one being executed at runtime depending on the result of the evaluation of the simplified test.

6.1 A Simple Example

We start by demonstrating the capacity to generate tight runtime tests using a simple example that corresponds to the code of X-MADD. As input to our toolchain, we provided the original code along with pragma(s) to specify the transformation for which a validity test is to be generated.

Here, we mark the inner-most loop for parallelization (e.g., for SIMD). This program is fed to ASTER, which produces the complete (with free-variables) system of constraints of the validity test. This system is then passed to our quantifier elimination tool, which after simplification successfully eliminates all existential variables (i.e., loop indices) and ends up generating the following runtime test:

```cpp
for (j = 0; j < K; j++)
#pragma parallel for
for (i = 0; i < J; i++)
S1: a[(N*(ai+1)+aj)+j] = \[\{\{N*(ai+1)+aj\}\}+j\]
+ \[((bj+bi)+j)\]
+ \[((bj+bi)+j)\]
+ \[((bj+bi)+j)\]
\]/4;
```

If at least one of the disjunction evaluates to true then the code branches to the original (non-optimized) version. Note that evaluating this test at runtime is extremely quick in comparison to the naive test produced by ASTER, thanks to the elimination of quantifiers.

For comparison purpose we have also evaluated QEPCAD-B [8] for the simplification process. Here QEPCAD-B determines the system to be false for the WAW dependence, but fails to process the systems for all remaining dependences.

6.2 Evaluation

We now demonstrate the ability for our technique to generate a low-overhead runtime validity test for complex polynomial array index functions that may alias. We also present experimental data about the performance improvements that can be reached by enabling aggressive loop transformations on these programs. We emphasize here that our purpose is not to find the utmost best transformation for each program, but instead to demonstrate the potential impact of implementing complex polyhedral transformations safely, after a complete may-alias and validity runtime check.

Experimental protocol. All experiments were conducted on a 4-core Intel Haswell Core i5-4590 CPU @ 3.30GHz running Fedora 25, with Turbo boost and DVFS/frequency scaling turned off. All benchmarks have been compiled with GCC 6.3.1 with flags `-O3 -fopenmp -march=native -mtune=native and ICC 17.0.2 with flags -03 -xHost -parallel -fopenmp. Each experiment was run 20 times, and we report the average performance. Note that the standard deviation of the execution time never exceed 2% in any of the experiments we conducted. We use Polybench/C 4.1 with DATA_TYPE_IS_DOUBLE and MEDIUM_DATASET.
Table 1: Results summary. Original, Optimized GF/s and Speed-up of the applications when compiled gcc or icc. If it was
various domains, including linear algebra, image processing, or
of the solved systems. The Transformation performed is reported, as described in Sec. 6.2.

The number of systems for which QEPCAD obtained a result, timed out or ended with error (out of memory). The Precision of the solved systems. The Transformation performed is reported, as described in Sec. 6.2.

<table>
<thead>
<tr>
<th>Appli. name</th>
<th>gcc – GF/s</th>
<th>icc – GF/s</th>
<th>Ex Op</th>
<th>#P</th>
<th>Naive test – O(n²)</th>
<th>Simplified test</th>
<th>QEPCAD-B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcio</td>
<td>opt</td>
<td>upx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>gcc</td>
<td>opt</td>
<td>upx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2mm</td>
<td>2.0</td>
<td>9.6</td>
<td>4.7</td>
<td>2.8</td>
<td>8.8</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>3mm</td>
<td>1.4</td>
<td>11.0</td>
<td>8.0</td>
<td>2.0</td>
<td>12.4</td>
<td>6.3</td>
<td></td>
</tr>
<tr>
<td>atax</td>
<td>1.8</td>
<td>5.8</td>
<td>3.2</td>
<td>8.0</td>
<td>5.4</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>binc</td>
<td>1.2</td>
<td>5.8</td>
<td>4.8</td>
<td>1.2</td>
<td>4.9</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>chol</td>
<td>4.3</td>
<td>4.3</td>
<td>1.0</td>
<td>6.4</td>
<td>6.4</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>correl</td>
<td>1.3</td>
<td>2.2</td>
<td>1.8</td>
<td>1.5</td>
<td>2.4</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>covari</td>
<td>1.3</td>
<td>12.1</td>
<td>9.5</td>
<td>1.5</td>
<td>16.0</td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>ddtge</td>
<td>1.8</td>
<td>9.0</td>
<td>5.0</td>
<td>2.4</td>
<td>11.7</td>
<td>4.8</td>
<td></td>
</tr>
<tr>
<td>FW</td>
<td>4.5</td>
<td>9.2</td>
<td>2.0</td>
<td>4.4</td>
<td>7.1</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>gmem</td>
<td>9.3</td>
<td>36.4</td>
<td>3.9</td>
<td>13.4</td>
<td>40.1</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>gemver</td>
<td>2.6</td>
<td>5.8</td>
<td>2.2</td>
<td>4.3</td>
<td>10.9</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td>gesumm</td>
<td>1.2</td>
<td>0.9</td>
<td>0.7</td>
<td>1.2</td>
<td>8.3</td>
<td>7.0</td>
<td></td>
</tr>
<tr>
<td>gamm</td>
<td>1.0</td>
<td>2.9</td>
<td>2.9</td>
<td>1.1</td>
<td>2.9</td>
<td>2.7</td>
<td></td>
</tr>
<tr>
<td>heat3D</td>
<td>11.7</td>
<td>26.9</td>
<td>2.3</td>
<td>10.1</td>
<td>31.9</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>jacoID</td>
<td>8.5</td>
<td>4.2</td>
<td>0.5</td>
<td>12.7</td>
<td>4.2</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>jaco2D</td>
<td>8.0</td>
<td>11.4</td>
<td>1.4</td>
<td>3.6</td>
<td>13.1</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>lu</td>
<td>1.3</td>
<td>1.3</td>
<td>1.0</td>
<td>2.7</td>
<td>2.8</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>mvt</td>
<td>1.2</td>
<td>4.3</td>
<td>3.5</td>
<td>2.5</td>
<td>4.1</td>
<td>1.6</td>
<td></td>
</tr>
<tr>
<td>sdeid</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
<td>1.1</td>
<td>1.1</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>symm</td>
<td>2.9</td>
<td>3.6</td>
<td>1.2</td>
<td>2.7</td>
<td>3.3</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>sty2k</td>
<td>1.6</td>
<td>2.2</td>
<td>1.3</td>
<td>2.0</td>
<td>2.4</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>syrk</td>
<td>5.1</td>
<td>25.9</td>
<td>5.1</td>
<td>6.0</td>
<td>21.5</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>trisol</td>
<td>1.0</td>
<td>1.2</td>
<td>1.2</td>
<td>1.6</td>
<td>8.0</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>trimm</td>
<td>1.6</td>
<td>1.9</td>
<td>1.2</td>
<td>2.1</td>
<td>1.6</td>
<td>0.8</td>
<td></td>
</tr>
</tbody>
</table>

Note that codes with polynomial (e.g., linearized) array accesses do not arise only from compiler IR lowering. For example Netlib BLAS kernels on symmetric matrices, e.g. in [37], use compacted storage leading to polynomial access functions. We leave the evaluation of such codes for future work, as our current prototype does not support memory address registers, used to compute an array displacement that depends on the loop induction variable.

Candidate optimization generation. We have generated one candidate optimization per benchmark. This optimization is the result of a complex polyhedral transformation implemented by the PoCC compiler [47]. We have extended PoCC to output, in addition to the transformed program, a series of pragmas describing the transformation applied. ASTER reads this information to build the initial validity test as described in Sec. 4, before we perform quantifier elimination.

The optimization candidate was generated using high-level PoCC optimization primitives, such as the PLuTo optimizer [5] which computes a sequence of loop transformations to enable parallelism and data locality enhancements, typically using loop tiling. This transformation, taking the form of a multidimensional schedule for each statement, can be seen as a sequence of high-level loop transformations. We report in Table 1, last column, the transformation as a sequence of fusion, Fission, loop interchange, thread-level parallelism, skewing, tiling, vectorization.

We recall that our objective is not to demonstrate the maximal performance achievable for each benchmark, but only to demonstrate the feasibility and potential impact of using polyhedral transformations on non-affine programs. For example, we have not tuned the tile sizes, or did not perform extensive exploration of the performance of various possible polyhedral optimizations. We mostly followed the data from Park et al. [45] which points to good PoCC optimizations for PolyBench.
Comparison with QEPCAD-B. We compared our approach against QEPCAD-B version b 1.69 [8], a publicly available quantifier elimination implementation that uses Partial Cylindrical Algebraic Decomposition. As most of our systems contain more than eight variables, we allow QEPCAD-B to execute with the maximum amount of memory cells we could use (by using -N1073741822). As, QEPCAD-B turns out to be quite slow, the overall number of systems being close to 2000, we limited the execution time per system to 30 minutes. However, as we will see further, most of the time QEPCAD-B gives up, running out of memory.

Detailed results. Table 1 presents the results on PolyBench/C 4.1, using the medium dataset size where programs typically run in a few milliseconds, to exacerbate any potential effect of executing the runtime test.4 We report for each case statistics on the size of the runtime test, before (naive) and after (simplified) applying our quantifier elimination strategy. Note that in all cases, the simplified test has a complexity of $O(1)$, meaning no loop is occurring in the simplified test. The runtime test computation compared to the total region execution time is negligible. It amounts here to evaluating a single large conditional, with no loop in it. In our experiments, we measured this overhead and confirmed it is marginal, typically well below 0.01% of the program execution time.

We also present results using a variation of PolyBench’s Floyd-Warshall benchmark, where we use a compact (triangular) representation. No existing polyhedral technique, including over-approximation, could generate a tiling-based optimization due to the purely polynomial access. With our proposed scheme, off-the-shelf polyhedral optimizers can be seamlessly applied on programs with polynomial array access functions, leading to potentially high gains as shown in Table 1.

Precision. Elimination techniques could generate correct but very conservative solutions, leading to a condition that always evaluates to true even if the transformation is in fact safe. In our experiments we compare our results to QEPCAD-B. The numerous failures (that is, a value $> 0$ in the #to and #err columns) illustrate the limitations of using QEPCAD-B in this context. The advantage of our FML scheme is that it allows the use of cutting planes on-demand, allowing to divide the space into different systems that can be processed in parallel and have their resulting systems recombined when all quantifiers have been eliminated without lost of precision. To evaluate the precision of our technique, we ask for sophisticated transformations to be done by the optimizer, including vectorization, tiling and parallelization, leading to complex systems to be simplified.

As we can see in Table 1, despite the high complexity of the initial systems in our experiments, our scheme leads to a test that turns out to be tight, allowing the use of the optimized code in all cases.5 Our FML scheme successfully processed all systems within the time limit, while QEPCAD-B only successfully produced a final result to jacobi-1d, mvt and trisolv applications. For all remaining tests it timed out or failed (running out of memory) to generate the quantifier free formula.6 To evaluate the precision of QEPCAD-B, we replaced all system that it failed to solve with our (precise) solution, so as to build a complete test. For six benchmarks (bigc, correlation, covariance, FW, gemm, gemver) the test fails to validate the transformation at runtime showing that for each of those benchmarks, at least one of the system generated by QEPCAD-B is not precise enough.

Focusing on the Floyd-Warshall algorithm, for a symmetric distance graph stored in a packed lower triangular matrix we obtain four different loops, depending on the size relations between $i, j, k$. In this example we described a distinct optimization per loop nest, as the following:

```c
void kernel_floyd_warshall_tri (int n, double *d);
int i, j, k, I, J, K;
#pragma scop
for (k) /\ 0 <= k < i < j < n
  for (parallel (i))
    for (j)
      s1: d[i*(i-1)/2+j] = d[i*(i-1)/2+k] + d[j*(j-1)/2+k];
      for (k) // \ 0 <= j < k < i < n
        for (parallel (i))
          for (j)
            s2: d[i*(i-1)/2+j] = d[i*(i-1)/2+k] + d[k*(k-1)/2+j];
            for (k) /\ 0 <= k < j < k < n
              for (parallel (i))
                for (j)
                  s3: d[i*(i-1)/2+j] = d[k*(k-1)/2+j] + d[k*(k-1)/2+j];
                    for (k, step = 32) // \ 0 <= i < j < k < n
                      for (parallel (i))
                        for (j)
                          s4: d[j*(j-1)/2+k] = d[k*(k-1)/2+j] + d[j*(j-1)/2+k];
```
checked before executing the optimized code in an inspector/executor manner. In the approaches advocated by Venkat et al. [58], the inspector corresponds to loops that actually inspect the data (i.e., the test complexity is not $O(1)$). This approach is motivated as it focuses on indirect indexes through read-only indexing arrays. Hybrid analysis with $O(1)$ runtime checks is used in [1, 27, 34, 51, 53] but it is always restricted to testing non-overlapping of intervals. This is useful in testing if a loop is fully parallel (or could be vectorized) but cannot serve as testing the validity of arbitrary loop transformation (such as loop-interchange). Hence, [1] only tackles the may-aliasing problem while GCC is capable of performing auto-vectorization [34] using aliasing and interval overlap tests. Rus et al. [53], and later Oancea and Rauchwerger [44] use “uniform set representation” along with appropriate inference rules to reason about data dependence, and an associated predicate language for efficient runtime tests that validate loop parallelization.

**Array Delinearization.** Another topic that is related to our work is array delinearization, where a set of loop bounds and polynomial access functions is used to reconstruct array dimensions and sizes.

As far as we know, the oldest attempt at delinearization dates back to the work of Maslov [42], but recent advances have made it applicable to linear access functions with parametric coefficients [24, 25]. Whereas delinearization consists in reconstructing affine array accesses to compute or apply a transformation, our work consists in testing the validity of a given transformation directly on the linearized accesses. However, apart the fact that this approach does not tackle pointer disambiguation, this pragmatic approach success rate is related to its “library” of access patterns. In particular, array delinearization in [25] does not handle arbitrary polynomials (e.g., for triangular array accesses), and requires some assumptions on the possible sizes of rectangular arrays.

**Handling Polynomials & Quantifier Elimination.** Polynomials have long been recognized as an important class of functions for loop analysis. Clauss [12] uses Bernstein basis decomposition to maximize a polynomial over a polyhedron. Unfortunately this technique is restricted to real numbers and its tentative use to design a quantifier elimination scheme leads to too imprecise results. Recently, Feautrier [19] has drawn the attention on two theorems on polynomials (see Section 5), and listed several potential applications. Our work can be seen as leveraging these theorems for the case of variable elimination.

The work that appears nearest to ours is Größlinger’s treatment of quantifier elimination [26, 27]. Constraint system splitting is used to structure the final projection as a tree, and no further simplification seems to be applied to the result. Another difference is that Fourier-Motzkin is applied only to polynomials that are linear in loop counters but have parametric coefficients, i.e., it could not be applied to our triangular array example. Several applications of the technique are mentioned, but as far as we know the resulting algorithm has never been applied to data dependence analysis. We use QEPCAD-b [9] to compare our results against cylindrical algebraic approaches, as used in that work. Whenever a solution could be provided (i.e. eliminate all variables leading to a $O(1)$ test) the result was not precise enough, prohibiting the use of the optimized code. More recently, on the work of Suriana [56], an extension to the FME was developed to handle parameter coefficients and used to produce loop boundaries. Limited to systems with a single symbolic coefficient, the author performs normalization as described on [50]. Without a technique to determine the sign of parametric coefficient the scheme always evaluate all possibilities, except when determined in the environment cache.

8 CONCLUSION AND FUTURE WORK

This paper developed a dynamic disambiguation technique to address the issue of dependence analysis and program optimization validity, suitable for many loop transformations such as skewing, tiling, vectorization, interchange, loop invariant code motion, parallelization, etc. We particularly focused on polynomial access functions, which are a challenge for current compiler analyses.

8.1 New approaches and techniques

From the applications described in Section 3, we observed that many techniques are related to our work, either directly or by the fact that they consider very similar problems. Such techniques include loop bounds computation, symbolic execution, and so on. Among the techniques described in this section, we include:

- **Feautrier’s theorems**: These theorems have been used to develop an improved method for loop bounds computation.
- **Cylindrical algebraic decompositions**: These techniques have been used to improve the precision of the results.
- **Quantifier elimination techniques**: These techniques have been used to optimize the code.

Future work will involve investigating the potential of these techniques in the context of data dependence analysis and program optimization.