Simplification and Run-time Resolution of Data Dependence Constraints for Loop Transformations

Diogo Sampaio, Louis-Noël Pouchet, Fabrice Rastello

Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations

Diogo N. Sampaio
Univ. Grenoble Alpes, Inria, CNRS,
LIG F-38000
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,
LIG F-38000
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 dependencies 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 [57]), 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):

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    S(i,j);
```

Suppose we want to permute the two loops:

```
for (j = 0; j < n; j++)
  for (i = 0; i < n; i++)
    S(i,j);
```

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:

```
valid=true
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (i' = 0; i' < n; i'++)
      for (j' = 0; j' < n; j'++)
        if (f(i,j) = f(i',j') && (i,j) <lex (i',j') && T(i,j) ≮lex T(i',j'))
          valid=false
```

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 dependencies, 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 xMMADD. 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]).

```
void xMMADD (int M, int N, int K, double A[M][N],
    int ai, int aj, double B[M][N], int bi, int bj)
  for (i = 0; i < K; i++)
    for (j = 0; j < K; j++)
      A[ai+i][aj+j] += (B[bi+i-1][bj+j]+B[bi+i+1][bj+j]
                      +  B[bi+i][bj+j-1]+B[bi+i][bj+j+1])/4
```

In this example, there is no `restrict` 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):

```
for (i = 0; i < K; i++)
  if( (A+(ai+i)*N+aj << 2) == (B+(bi+i)*N+bj << 2) ||
      (A+(ai+i)*N+aj << 2) == (B+(bi+i-1)*N+bj << 2)||
      (A+(ai+i)*N+aj << 2) == (B+(bi+i+1)*N+bj << 2) )
    for (j = 0; j < K; j++)
      ... // original version
  else
    for (j = 0; j < K; j+=4)
      ... // vectorized version
```

[1] Note 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[ai+i][aj+j+ 0 .. 3]` and `B[bi+i][bj+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[i,j]` is represented as `T[i*(i-1)/2+j]` if $i \geq j$ 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 computation of Floyd-Warshall with symmetries uses this representation. It turns out that such code would highly benefit from tiling [60], as the data reuse availble 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 `t[H][H]` is written as `t[H*j+i]`. Multi-dimensional array linearization is widely present in high-performance legacy code. One of the reason is that languages such as C89 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 (i = 1; i < m; i++)
  for (j = 1; j < n; j++)
    t[H*j+i] = t[-H+H*j+i] + t[-1-H+H*j+i];
```

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 `m` 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).

```
for (k = 2; k < n; k++) // j<i<k
  for (i = 1; i < k; i++)
    for (j = 0; j < i; j++)
      S1: d[i*(i-1)/2+j] min= d[k*(k-1)/2+i]+d[k*(k-1)/2+j];
for (k = 1; k < n-1; k++) // j<k<i
  for (i = k+1; i < n; i++)
    for (j = 0; j < k; j++)
      S2: d[i*(i-1)/2+j] min= d[i*(i-1)/2+k]+d[k*(k-1)/2+j];
```

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:

```
for (t = 0; t < 2; t++)
  if (t == 0)
    for (k = 2; k < n; k++) // j<i<k
      for (i = 1; i < k; i++)
        for (j = 0; j < i; j++)
          S1: ...
  else if (t == 1)
    for (k = 1; k < n-1; k++) // j<k<i
      for (i = k+1; i < n; i++)
        for (j = 0; j < k; j++)
          S2: ...
```

In this form any statement is indexed by a vector of possibly larger size than the original loop depth ((t, 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 $\mathcal{D}_{S1} = \{(t, k, i, j),\ t = 0 \wedge 0 \leq j < i < k < n\}$. 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 T_{S1} for statement $S1$) from $\mathcal{D}$ to $\mathcal{D}' \in \mathbb{Z}^s$ a vector space of dimension $s \geq 1$ ($\mathbb{Z}$ represents the integers). $T_{S1}(0, k, i, j) = T_{S1}(0, k', i', j')$ means that $S1_{0,k,i,j}$ and $S1_{0,k',i',j'}$ are executed in parallel. $T_{S1}(0, k, i, j) \prec_{lex} T_{S1}(0, k', i', j')$ means that $S1_{0,k,i,j}$ is executed before $S1_{0,k',i',j'}$[2]. 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 $T_{s1}(0, k, i, j) = (0, k, j, i)$. The canonical form allows to express the schedule of the original code as the identity function from $\mathcal{D}$ to $\mathcal{D}$.

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 $S1_{0,k,i,j}$ writes to location $W_1(0, k, i, j) = T + i(i-1)/2 + j$ 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 $S1_{k,i,j}$ and $S1_{k',i',j'}$ from W_1 to R_2 as soon as: (1) $(0, k, i, j) \in \mathcal{D}$ strictly precedes $(0, k', i', j') \in \mathcal{D}$ in the original schedule; which is written as $(0, k, i, j) \prec_{lex} (0, k', i', j')$ ($\prec_{lex}$ represents a lexicographic 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, $S1_{0,3,2,1} <_d S1_{0,4,2,1}$ 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-write is an output dependence; A write-after-read is an anti-dependence.

[2] $(a_1, \ldots, a_n) \prec_{lex} (b_1, \ldots, b_m)$ iff there exists an integer $1 \leq i \leq min(n, m)$ s.t. $(a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1})$ and $a_i < b_i$.

For any schedule to be *valid* it must preserve all such dependences. As an example, for the previously considered loop interchange $T_{S1}(0,k,i,j) = (0,k,j,i)$, the dependence $S1_{0,3,2,1} \prec_d S1_{0,4,2,1}$ is preserved as $T_{s1}(0,3,2,1) = (0,3,1,2) \prec_{lex} T_{s1}(0,4,2,1) = (0,4,1,2)$. 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.

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

$$\begin{array}{lll} & v \in \mathcal{D}_A \wedge v' \in \mathcal{D}_{A'} & \text{(domain)} \\ \wedge & v \prec_{lex} v' & \text{(orig. sched.)} \\ \wedge & A(v) = A'(v') & \text{(same access loc.)} \\ \wedge & T_A(v) \not\prec_{lex} T_{A'}(v') & \text{(violation)} \end{array}$$

where: **(1)** s is the dimension of the loop nest in canonical form; **(2)** A and A' are access functions (e.g. W_1 and R_2 in the example of Section 3), one of them necessarily being a write access (to generate a may-dependence); **(3)** $\mathcal{D}_A$ (resp. $\mathcal{D}_{A'}$) is the set of points in the domain where A (resp. A') is accessed ($\mathcal{D}_{S1}$ for W_1 and R_2 in our example); **(4)** mD represents all possible pairs of such functions that could lead to a may-dependence (a dependence the compiler cannot disambiguate statically); **(5)** T_A (resp. $T_{A'}$) represents the schedule of the statement including access A (resp. A') after transformation (T_{S1} 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 safe transformation. In that sens, 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 $(A, A') = (R_2, W_1)$, where $A(i,j) = R_2(i,j) = -H + Hj + i$ and $A'(i,j) = W_1 = Hj + i$. Consider loop interchange as the desired transformation, i.e. $T(i,j) = (j,i)$. The system of equations writes as follow:

$$\begin{array}{lll} \exists (i,j,i',j') \in \mathbb{Z}^4 \text{ s.t.} & & \\ & 1 \le i < m \wedge 1 \le i' < m & \text{(domain)} \\ \wedge & 1 \le j < n \wedge 1 \le j' < n & \text{(domain)} \\ \wedge & (i,j) \prec_{lex} (i',j') & \text{(orig. sched.)} \\ \wedge & -H + Hj + i = Hj' + i' & \text{(same access loc.)} \\ \wedge & (j,i) \not\prec_{lex} (j',i') & \text{(violation)} \end{array}$$

$(i,j) \prec_{lex} (i',j')$ writes as $i < i' \vee (i = i' \wedge j < j')$ and thus (orig. sched.) $\wedge$ (violation) simplifies to: $i < i' \wedge j' < j$.

Skewing such as $T(i,j) = (i, i+j)$ can be used to enable interchange and tiling. Combined with an interchange, (violation) writes as $(i+j, i) \not\prec_{lex} (i'+j', i')$.

Parallelization is valid if there are no loop-carried dependencies. To illustrate this point we consider ADI with the above skewing $T(i,j) = (i+j, j)$ that exposes parallelism along the innermost loop. Expressing that we want this loop to be parallel simply consists in folding it, getting the schedule $T(i,j) = (i+j)$. In that case the system of inequalities (for may-dependence (R_2, W_1)) simply writes as follows:

$$\begin{array}{lll} \exists (i,j,i',j') \in \mathbb{Z}^4 \text{ s.t.} & & \\ & 1 \le i < m \wedge 1 \le i' < m & \text{(domain)} \\ \wedge & 1 \le j < n \wedge 1 \le j' < n & \text{(domain)} \\ \wedge & (i,j) \prec_{lex} (i',j') & \text{(orig. schedule)} \\ \wedge & -H + Hj + i = Hj' + i' & \text{(same access loc.)} \\ \wedge & i + j \not\prec_{lex} i' + j' & \text{(violation)} \end{array}$$

Loop Invariant Code Motion means that (in terms of dependence) the hoisted code does not depend on any computation inside the loop. Consider the statement `S1` in the example of Section 3. The load `d[k*(k-1)/2+i]` can be hoisted outside the inner most loop `j`. Considering may-anti-dependence with the write operation at the same statement `d[i*(i-1)/2+j]`, we get the following system of inequalities:

$$\begin{array}{lll} \exists (i,j,k,i',j',k') \in \mathbb{Z}^6 \text{ s.t.} & & \\ & 2 \le k < n \wedge 2 \le k' < n & \text{(domain)} \\ \wedge & 1 \le i < k \wedge 1 \le i' < k' & \text{(domain)} \\ \wedge & 0 \le j < i \wedge 0 \le j' < i' & \text{(domain)} \\ \wedge & (k,i,j) \preceq_{lex} (k',i',j') & \text{(orig. schedule)} \\ \wedge & i(i-1)/2 + j = k(k-1)/2 + i & \text{(same access loc.)} \\ \wedge & (k,i,1,j) \not\prec_{lex} (k',i',0,j') & \text{(violation)} \end{array}$$

Vectorization. Consider again the ADI case after skewing ($T(i,j) = (i+j, j)$). There exists two ways to test for vectorization. The first and preciser way to express the condition to vectorize the new inner-most loop by a vector of size l is to use: $T(i,j) = (i+j, \lfloor j/l \rfloor)$. In that condition the system of equations for (R_2, W_1) may-dependence writes:

$$\begin{array}{lll} \exists (i,j,i',j') \in \mathbb{Z}^4 \text{ s.t.} & & \\ & 1 \le i < m \wedge 1 \le i' < m & \text{(domain)} \\ \wedge & 1 \le j < n \wedge 1 \le j' < n & \text{(domain)} \\ \wedge & (i,j) \prec_{lex} (i',j') & \text{(orig. schedule)} \\ \wedge & -H + Hj + i = Hj' + i' & \text{(same access loc.)} \\ \wedge & (i+j, \lfloor j/l \rfloor) \not\prec_{lex} (i'+j', \lfloor j'/l \rfloor) & \text{(violation)} \end{array}$$

Note that floor (or modulo) can be handled by adding two new variables (say r and d): $\lfloor j/l \rfloor = d \wedge 0 \le r < l \wedge dl + r = j$. Alternatively the (violation) condition (knowing that $i \le i'$ from the (orig. schedule) condition) can also be relaxed into $j' - j < l \wedge i + j \ge i' + j'$. This last condition expresses standard test that enforces any dependence to be of length bigger than l for vectorization to be valid. Notice also that this test is more precise than the standard non-aliasing test [4, p. 64-67] which for the simple dot-product benchmark would consider vectorization not to be possible if one of the source and the destination vectors are exactly the same. Here, the strict inequality in the (orig. schedule) constraint allows to optimize this case.

Loop-Body Scheduling / Software Pipelining. As observed above, the necessary condition for vectorization of size l to be valid is that all loop-carried dependences must be of distance greater or equal to l. 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 $S1$. The band (k,i,j) of the domain $D_{S1} = \{(0,k,i,j),\ 0 \leq j < i < k < n\}$ is fully permutable if individually (k,i), (k,j), and (i,j) can be interchanged [48]. The conditions can actually be expressed in a more condensed way [2]: all dependencies should be within the positive cone. Restricting to may-dependence (W_1, R_3) $W_1(0,k,i,j) = i(i-1)/2+j, R_3(0,k,i,j) = k(k-1)/2+i$, the system of inequalities is written as follows:

$$
\begin{array}{lll}
\exists (t,k,i,j,t',k',i',j') \in \mathbb{Z}^8 \text{ s.t.} & & \\
\quad t = 0 \leq j < i < k < n & & \text{(domain)} \\
\quad t' = 0 \leq j' < i' < k' < n & & \text{(domain)} \\
\wedge \quad (t,k,i,j) \prec_{lex} (t',k',i',j') & & \text{(orig. sched.)} \\
\wedge \quad i(i-1)/2 + j = k'(k'-1)/2 + i' & & \text{(access loc.)} \\
\wedge \quad (k,i,j) \not\leq (k',i',j') & & \text{(violation)}
\end{array}
$$

where $(k,i,j) \leq (k',i',j')$ writes as $k \leq k' \wedge i \leq i' \wedge j \leq j'$.

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:

$$T(0,k,i,j) \not\prec_{lex} T(0,k',i',j') \vee T(0,k,i,j) \not\leq T(0,k',i',j')$$

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/retiming, 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 $\mathcal{S}(P,V) : [\{E_0, E_1 \cdots E_n\} \geq 0] \wedge \left[\{E'_0, E'_1 \cdots E'_m\} = 0\right]$ 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 $\mathcal{S}(P,V)$ into $\mathcal{S}'(P)$.

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 $\mathcal{S} : \{E_1, E_2 \ldots E_n\} \geq 0$, Schweighofer provides a method to evaluate if the inequality $\mathcal{I} : E \geq 0$ is implied by $\mathcal{S}$. If $\mathcal{I}$ can be written as the sum of the product of any power of the inequalities of $\mathcal{S} \cup \{1\}$ multiplied by non-negative factors (multivariate-polynomial in $E_1, \ldots E_n$ with positive coefficients), then $\mathcal{S}$ implies $\mathcal{I}$. As an example, if the system $S = \{E_1 = -xy - x \geq 0 \wedge E_2 = x - 1 \geq 0 \wedge E_3 = y \geq 0\}$ is true, then $y \geq 1$. Indeed, $y - 1 = 2E_3 + E_2 + E_1 + E_2 \cdot E_3$. Unfortunately, raising a subset $\mathcal{S}' \subset \mathcal{S}$ of cardinality m to degree d, leads to a polynomial (in E_i) with a number of terms combinatorial with m and d. Take as an example the subset $\mathcal{S}' = \{E_1, E_2\}$ (we have $m = 2$) and $d = 2$. The goal (to prove the positiveness of E) is to find coefficients λ_i of the polynomial $\lambda_0 + \lambda_1 E_1 + \lambda_2 E_2 + \lambda_{12} E_1 \cdot E_2 - E = 0$ 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 $(\lambda_0 + 1) + (-\lambda_1 + \lambda_2 + \lambda_{12})x - y + (-\lambda_1 + \lambda_{12})xy - \lambda_{12}x^2 - \lambda_{12}x^2 y = 0$ leads to the linear program $\{\lambda_i \geq 0 \wedge \lambda_0 + 1 = 0 \wedge -\lambda_1 + \lambda_2 + \lambda_{12} = 0 \wedge -1 = 0 \wedge \ldots\}$ that has no solution in that case. Our implementation chooses, heuristically, subsets $\mathcal{S}' \subset \mathcal{S}$ of bounded cardinality m, and limits the maximum degree d to which $\mathcal{S}'$ are raised. Solving the LP is heuristically done by using a linear solver where positiveness of the λ_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 λ_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 $\mathcal{S}$ implies $E \geq 0$ or $\mathcal{S}$ implies $-E \geq 0$. When both are implied, if one of the inequalities can be made strict, then $\mathcal{S} \equiv$ false, otherwise $E = 0$. Whenever none of the inequalities can be proven to be implied by any $\mathcal{S}'$, 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 $\mathcal{S}$, the

[3] We actually use the technique in a more general context where we do not enforce variables to live in a compact polytope.

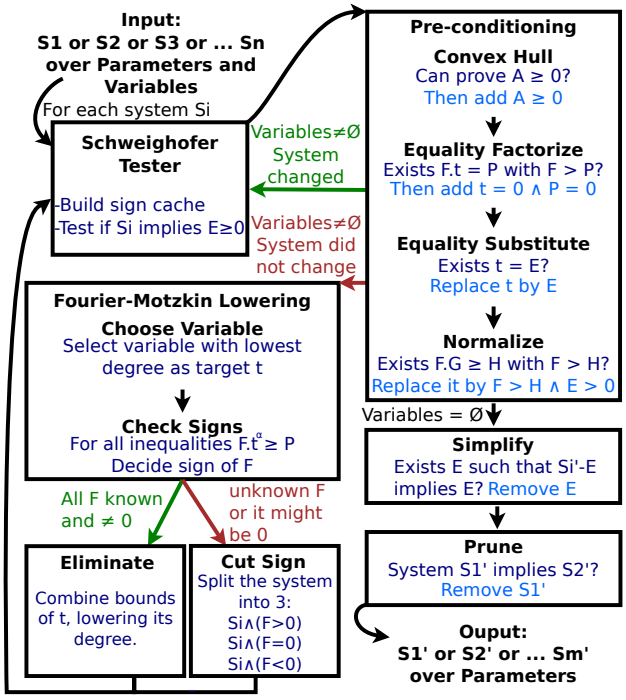

Figure 1: Quantifier Elimination scheme. Input and output are sets of conjunctive systems of polynomial inequalities. "A" stands for affine hyper-planes.

FME works as follows: (1) Consider the variable $t \in V$ and a set of inequalities $\mathcal{S}$ on that variable; (2) isolate t so that any inequalities from $\mathcal{S}$ can either be written as:

$$\begin{array}{lll} & t & \leq U_i(P, V \setminus \{t\}) \quad \text{(upper bounds)} \\ \text{or} \quad L_j(P, V \setminus \{t\}) \leq & t & \text{(lower bounds)} \end{array}.$$

(3) eliminate t from the system by combining each lower-bound with each upper-bound:

$$\forall (i,j), \quad L_j(P, V \setminus \{t\}) \leq U_i(P, V \setminus \{t\})$$

The basic FME is designed for a system made up of linear inequalities on $\mathbb{R}$. As we will see in Section 7 there exists several extensions. One considers linear inequalities but possibly with parameters as coefficients [27]. Another one considers linear inequalities on $\mathbb{Z}$. The goal here, is to extend the method to polynomial inequalities on $\mathbb{Z}$. The main difference with a linear system is that each elimination step (of the Convergence mode – see below) actually consists in **lowering** the degree of the variable t. The first step, Choose Variable, selects the variable t to eliminate, as the one with lowest degree. The second step, Check Signs has two modes. In the Precision mode, it writes every inequalities as:

$$\begin{array}{lll} & C_i(P,V).t & \leq U_i(P, V \setminus \{t\}) \\ \text{or} \quad L_j(P, V \setminus \{t\}) \leq & C_j(P,V).t & \end{array}$$

where each L_j and U_j do not depend on t. In the Convergence mode, it writes every inequalities as:

$$\begin{array}{lll} & C_i(P, V \setminus \{t\}).t^\alpha & \leq U_i(P,V) \\ \text{or} \quad L_j(P,V) \leq & C_j(P, V \setminus \{t\}).t^\alpha & \end{array}$$

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

$$\begin{array}{lll} & E(P,V) & \leq U'_i(P,V) \\ \text{or} \quad L'_j(P,V) \leq & E(P,V) & \end{array}$$

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_*. 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:

$$\begin{array}{ll} \mathcal{S}_{>0}: & \mathcal{S} \wedge (C_k > 0) \\ \mathcal{S}_{=0}: & \mathcal{S} \wedge (C_k = 0) \\ \mathcal{S}_{<0}: & \mathcal{S} \wedge (C_k < 0) \end{array}$$

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 $\mathcal{S}'$ the new system obtained from $\mathcal{S}$ after a lowering step. Obviously $\mathcal{S}'$ contains all the combined constraints $L'_j \leq U'_i$. But there is a subtlety concerning the constraints from $\mathcal{S}$ reported in $\mathcal{S}'$. In Precision mode, all constraints from $\mathcal{S}$ that do not increase the t-degree of $\mathcal{S}'$ are reported in $\mathcal{S}'$. In Convergence mode, all constraints from $\mathcal{S}$ with t-degree smaller than α are reported in $\mathcal{S}'$.

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 $v \geq c$, or $-v \geq c$) can be proven to be implied by $\mathcal{S}$, then this inequality is added to the system. This aids the Schweighofer 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 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 $E(P,V) \geq 0 \wedge E(P,V) \leq 0$.

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

elimination. Our normalization extends the one from [51] to non-numeric coefficients. For any inequality in the form $F(P)\cdot G(P,V) \geq H(P)$, if Schweighofer Tester proves that $|F(P)| \geq |H(P)|$, 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 $\mathcal{I} : E \geq 0$ we test if $\{\mathcal{S} \setminus \mathcal{I}\}$ implies $\mathcal{I}$. If it does then $\mathcal{I}$ is removed; If it can be proved that $\{\mathcal{S} \setminus \mathcal{I}\}$ implies $E < 0$, then a contradiction $(\mathcal{I} \wedge \neg\mathcal{I})$ is found, and the system is turned into false.

Prune. The last "simplification" step acts over the disjunction formed by all systems $\mathcal{S}'_n(P)$. Using the Schweighofer Tester, if a system $\mathcal{S}'_1$ implies all inequalities of a system $\mathcal{S}'_2$, then $\mathcal{S}'_1$ 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 xMMADD. 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.

```
for (i = 0; i < K; i++)
   #pragma parallel for
   for (j = 0; j < K; j++)
      S1: A[N*(ai+i)+aj+j] = (B[N*(bi+i-1)+bj+j]
                  + B[N*(bi+i+1)+bj+j]
                  + B[N*(bi+i)+bj+j-1] + B[N*(bi+i)+bj+j+1])/4;
```

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:

```
if (((aj+A+ai*N≥B+N*bi+bj)&&(B+N*bi+bj+K≥aj+A+ai*N+2))‖
    ((aj+A+ai*N+K≥B+N*bi+bj)&&(B+N*bi+bj≥aj+A+ai*N+2))‖
    ((aj+A+ai*N+K≥B+N*bi+bj+2)&&(B+N*bi+bj≥aj+A+ai*N))‖
    ((B+N*bi+bj+K≥aj+A+ai*N)&&(aj+A+ai*N≥B+N*bi+bj+2))‖
    ((aj+N+A+ai*N≥B+N*bi+bj+1)&&(B+N*bi+bj+K≥aj+N+A+ai*N+1))‖
    ((aj+N+A+ai*N+K≥B+N*bi+bj+1)&&(B+N*bi+bj≥aj+N+A+ai*N+1))‖
    ((aj+A+ai*N+K≥B+N+N*bi+bj+1)&&(B+N+N*bi+bj≥aj+A+ai*N+1))‖
    ((B+N+N*bi+bj+K≥aj+A+ai*N+1)&&(aj+A+ai*N≥B+N+N*bi+bj+1))
   ) { original } else { optimized }
```

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 `-O3 -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 exceeded 2% in any of the experiments we conducted. We use Polybench/C 4.1 with `DATA_TYPE_IS_DOUBLE` and `MEDIUM_DATASET`.

Appli.	gcc — GF/s		Spd	icc — GF/s		Spd	Ex	#P	Naive test — $O(n^2)$			Simplified test			QEPCAD-B				Trans
name	ori	opt	up×	ori	opt	up×	Op	MA	Cmp	#Ineqs	#Sys	Cmp	#Ine	#Sys	#obt	#to	#err	Pre	
2mm	2.0	9.6	4.7	2.8	8.8	3.2	T	5	n^4	4080	265	1	3435	318	17	70	178	T	psv
3mm	1.4	11.0	8.0	2.0	12.4	6.3	T	7	n^6	3277	212	1	1892	224	14	70	128	T	fFipv
atax	1.8	5.8	3.2	8.0	5.4	0.7	T	4	n^4	382	34	1	73	17	21	2	11	T	fipv
bicg	1.2	5.8	4.8	1.2	4.9	4.0	T	5	n^4	524	45	1	110	27	30	6	9	F	Fpv
choles	4.3	4.3	1.0	6.4	6.4	1.0	T	1	n^6	408	28	1	1	1	23	2	3	T	v
correl	1.3	2.2	1.8	1.5	2.4	1.6	T	4	n^6	1835	148	1	560	70	58	22	68	F	fipsv
covari	1.3	12.1	9.5	1.5	16.0	10.7	T	3	n^6	1057	77	1	704	41	27	29	21	F	fipv
doitge	1.8	9.0	5.0	2.4	11.7	4.8	T	3	n^8	287	13	1	11	2	5	3	5	T	fv
FW	4.5	9.2	2.0	4.4	7.1	1.6	T	1	n^6	475	25	1	1	1	17	0	8	F	pt
gemm	9.3	36.4	3.9	13.4	40.1	3.0	T	3	n^6	722	45	1	269	34	10	9	26	T	Fipv
gemver	2.6	5.8	2.2	4.3	10.9	2.6	T	9	n^4	1886	155	1	89	30	135	10	10	F	Fipv
gesumm	1.2	0.9	0.7	1.2	8.3	7.0	T	5	n^4	521	48	1	74	25	47	1	0	F	Fpv
gramsc	1.0	2.9	2.9	1.1	2.9	2.7	T	3	n^6	91	7	1	49	11	2	1	4	T	Fpv
heat3D	11.7	26.9	2.3	10.1	31.9	3.2	T	2	n^8	208k	8k	1	1	1	-	-	-	-	ipsv
jaco1D	8.5	4.2	0.5	12.7	4.2	0.3	T	2	1	1	1	1	1	1	1	0	0	T	v
jaco2D	8.0	11.4	1.4	3.6	13.1	3.6	T	2	n^4	528	44	1	103	37	4	16	24	T	ptv
lu	1.3	1.3	1.0	2.7	2.8	1.0	T	1	n^6	376	22	1	1	1	7	4	11	T	v
mvt	1.2	4.3	3.5	2.5	4.1	1.6	T	5	n^4	534	46	1	79	27	46	0	0	T	Fpv
seidel	0.9	0.9	1.0	1.1	1.1	1.0	T	1	n^4	544	40	1	2	2	1	0	39	T	Fpv
symm	2.9	3.6	1.2	2.7	3.3	1.2	T	3	n^6	411	26	1	178	28	1	9	16	T	F
syr2k	1.6	2.2	1.3	2.0	2.4	1.2	T	3	n^6	1211	74	1	523	89	10	31	33	T	ipv
syrk	5.1	25.9	5.1	6.0	21.5	3.6	T	2	n^6	722	45	1	316	34	4	18	23	T	Fpv
trisol	1.0	1.2	1.2	1.6	8.0	4.9	T	3	n^4	41	4	1	1	1	4	0	0	T	t
trmm	1.6	1.9	1.2	2.1	1.6	0.8	T	2	n^6	628	38	1	2015	111	4	13	21	T	Fip
FW-tri	2.2	3.1	1.4	1.7	3.8	2.3	T	1	n^6	7240	385	1	1	1	0	78	307	-	ptv

Table 1: Results summary. Original, Optimized GF/s and Speed-up of the applications when compiled gcc or icc. If it was Executed the Optimized code (True/False). The number of Pointers in May-Alias. The Naive and Simplified columns show: the Complexity order of the test, the total number of Ineqalites in the test, and the number of Systems (conjunctions) in the test. 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.

Benchmarks. We focus our evaluation on the PolyBench/C 4.1 [49] benchmark suite. It contains a variety of numerical kernels from various domains, including linear algebra, image processing, or datamining. PolyBench natively contains affine program regions using multidimensional arrays. We transformed each such array access into a linearized form, e.g., `A[i][j]` for an array `A[N][N+1]` is rewritten as `A[i*(N+1)+j]`. Note that in practice many compilers, e.g., LLVM, will linearize the multidimensional array accesses leading to non-affine access functions, and preventing optimization with off-the-shelf polyhedral optimizers. In specific cases techniques like optimistic delinearization may be applied [25], yet issues such as array padding may quickly prevent the technique to be applied even on PolyBench [25]. We show that in contrast our approach can lead to systematic successful analysis and optimization of linearized PolyBench/C benchmarks.

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:

```
1  void kernel_floyd_warshall_tri (int n, double *d){
2  int i, j, k, I, J, K;
3  #pragma scop
4  for (k)//0 <= k < j < i < n
5   for parallel (i)
6    for (j)
7  S1: d[i*(i-1)/2+j] min= d[i*(i-1)/2+k]+d[j*(j-1)/2+k];
8  for (k) //0 <= j < k < i < n
9   for parallel (i)
10   for (j)
11 S2: d[i*(i-1)/2+j] min= d[i*(i-1)/2+k] + d[k*(k-1)/2+j];
12 for (k)//0 <= j < i < k < n
13  for parallel (i)
14   for (j)
15 S3: d[i*(i-1)/2+j] min= d[k*(k-1)/2+i] + d[k*(k-1)/2+j];
16 for (K, step = 32)//0 <= i < j < k < n
17  for (J, step = 32)
18   for (I, step = 32)
19    for(k)
20     for parallel (j)
21      for (i)
22 S4: d[j*(j-1)/2+i] min= d[k*(k-1)/2+i] + d[j*(j-1)/2+k];
```

A total of 385 systems were generated and our quantifier elimination evaluated all to false. QEPCAD-B fails to process any of the 385 systems given a 30 min limit per system.

7 RELATED WORK

Related work reported here concerns: 1. static/dynamic analysis, and speculative execution as techniques (similar to our) to overcome the limitations of static dependence analysis and poor alias information; 2. Array delinearization that recovers multidimensional data-structure sizes by a pattern matching technique; 3. The handling of polynomials and in particular quantifier elimination schemes.

Static/dynamic analysis and speculative execution. Most of the existing work which goal is to expose parallelism and locality on code with poor alias/dependence information exploits speculative execution framework. As opposed to the vast majority that are restricted to instruction level parallelism (e.g. [11, 15, 20, 28]), the work of Jimborean at al. [33] later improved by [41, 55] allows to perform arbitrary complex polyhedral transformations: The idea is to instrument memory accesses during execution; if a linear pattern is observed, then an optimizing transformation is applied, expecting the access pattern to hold. Without guaranties that the memory access prediction would hold, a roll-back to a safe point must be done if a misprediction is detected. The main limitation of this approach is the overhead due to memory instrumentation and backtracking support. Our work based on hybrid analysis allows to avoid speculation as the validity of the transformation is

[4] Note that 6 of the 30 PolyBench/C 4.1 codes are missing, as they are not supported by the version of PoCC we used.

[5] By design of our experiments, the transformation we apply is always valid, so an accurate-enough validity test should always lead to executing the transformed code.

[6] Due time constraints we did not evaluate heat-3d test with QEPCAD-B

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. Such functions arise typically with low-level IR (which are more and more common to address performance portability) but also in legacy code with pointers (e.g. C), linearized arrays, etc. To make the approach practical, the test to be evaluated at runtime to assess the validity of a transformed program must be as simple as possible. This objective was addressed by developing a powerful new quantifier elimination scheme on integer multivariate-polynomials, as prior works suffer from significant precision issues when dealing with quantifiers on $\mathbb{Z}$.

The quality of the presented quantifier elimination technique is very important to make this approach realistic. In particular, it must be precise enough such that the test succeeds in practical cases, and must lead to negligible overhead. The integer aspect makes the problem very challenging due to rounding errors that must be repeatedly removed. The precision and overhead has been evaluated on a set of 25 benchmarks using complex loop transformations.

Due to the combinatorial aspect of the elimination scheme, our current implementation is clearly not suited for Just-In-Time compilation. Fortunately, even if in theory the number of inequalities can square at every step of the Fourier-Motzkin scheme, practice for affine systems many generated terms are redundant. It turns out to be the same for polynomial systems, and the use of the Schweighofer redundancy detection technique allows to make the approach really practical, requiring a few minutes to solve all the systems of our 20+ benchmarks. However, we believe many improvements are possible to decrease the analysis time. One consists in eliminating false systems (when cutting), using lifting to generate over-approximations such as intervals or polyhedra. A second one consists in caching Schweighofer's redundancy results as it turns out identical systems appear many times in practice. Another one consists in (when available) using profile-based values. Such information could be used for three purposes: first, if profiled values state that a term turns out to be true in practice, as it is always safe to replace a term by true (which means transformation is not possible) it can be eliminated in any conjunctions. Second, interesting cuts mostly appear on actual dependences. Dependences observed by profiling (using shadow memory) can be used to add initial cuts. Lastly, "degraded" but faster techniques that use lifting or Bernstein expansion when it turns out to be accurate enough on the simple parts of the system. One could run degraded schemes first, check if the result is accurate enough with regard to profiled values, and only if it is not accurate, use a more aggressive (and costly) approach.

REFERENCES

[1] Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2015), OOPSLA 2015, ACM, pp. 589–606.

[2] Banerjee, U. *Dependence analysis*, vol. 3. Springer Science & Business Media, 1997.

[3] Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., and Bastoul, C. The polyhedral model is more widely applicable than you think. In *Compiler Construction* (2010), Springer, pp. 283–303.

[4] Bik, A. J. C. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[5] Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI'08, Tucson, AZ* (2008).

[6] Bondhugula, U. K. R. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

[7] Boost c++ libraries. http://www.boost.org/, 1999.

[8] Brown, C. W. QEPCAD B: A Program for Computing with Semi-algebraic Sets Using CADs. *SIGSAM Bull. 37*, 4 (Dec. 2003), 97–108.

[9] Brown, C. W. Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM Bull. 37*, 4 (Dec. 2003), 97–108.

[10] Bugnion, E., Liao, S.-W., Murphy, B., Amarasinghe, S., Anderson, J., Hall, M., and Lam, M. Maximizing multiprocessor performance with the suif compiler. *Computer 29* (1996).

[11] Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA* (2006), IEEE Computer Society, pp. 227–238.

[12] Clauss, P., Fernandez, F., Garbervetsky, D., and Verdoolaege, S. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 17*, 8 (Aug 2009), 983–996.

[13] Collard, J.-F., Barthou, D., and Feautrier, P. Fuzzy array dataflow analysis. *ACM SIGPLAN Notices 30*, 8 (1995), 92–101.

[14] Colt. http://dst.lbl.gov/ACSSoftware/colt/, 1999.

[15] Da Silva, J., and Steffan, J. G. A probabilistic pointer analysis for speculative optimizations. In *ASPLOS* (2006), ACM, pp. 416–425.

[16] Efficient java matrix library. http://ejml.org.

[17] Feautrier, P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*, 1 (1991), 23–53.

[18] Feautrier, P. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming 21*, 6 (1992), 389–420.

[19] Feautrier, P. The power of polynomials. In *5th International Workshop on Polyhedral Compilation Techniques* (2015). http://impact.gforge.inria.fr/impact2015/.

[20] Fernández, M., and Espasa, R. Speculative alias analysis for executable code. In *PACT* (2002), IEEE, pp. 222–231.

[21] Gimple. https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[22] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., and Temam, O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming 34*, 3 (2006), 261–317.

[23] Grosser, T., Groesslinger, A., and Lengauer, C. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters 04*, 22 (2012), X.

[24] Grosser, T., Pop, S., Ramanujam, J., and Sadayappan, P. On recovering multi-dimensional arrays in polly. In *5th International Workshop on Polyhedral Compilation Techniques* (2015). http://impact.gforge.inria.fr/impact2015/.

[25] Grosser, T., Ramanujam, J., Pouchet, L.-N., Sadayappan, P., and Pop, S. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (New York, NY, USA, 2015), ICS '15, ACM, pp. 351–360.

[26] Grösslinger, A. Extending the polyhedron model to inequality systems with non-linear parameters using quantifier elimination. Master's thesis, Department of Informatics and Mathematics, University of Passau, 2003.

[27] Grösslinger, A., Griebl, M., and Lengauer, C. Quantifier elimination in automatic loop parallelization. *Journal of Symbolic Computation 41*, 11 (2006), 1206 – 1221. Special Issue on the Occasion of Volker Weispfenning's 60th BirthdaySpecial Issue on the Occasion of Volker Weispfenning's 60th Birthday.

[28] Huang, A. S., Slavenburg, G., and Shen, J. P. Speculative disambiguation: a compilation technique for dynamic memory disambiguation. In *ISCA* (1994), pp. 200–210.

[29] IBM. XL C/C++ for Linux. http://www-03.ibm.com/software/products/en/xlcpp-linux.

[30] Intel. Math kernel library. https://software.intel.com/en-us/intel-mkl, 2003.

[31] Intel, Willow Garage, and Itseez. http://opencv.org/.

[32] Irigoin, F., and Triolet, R. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1988), ACM, pp. 319–329.

[33] Jimborean, A. *Adapting the Polytope Model for Dynamic and Speculative Parallelization*. Theses, Université de Strasbourg, Sept. 2012.

[34] Kaplansky, V. Gcc patch: implementation of alias versioning in vectorizer. https://gcc.gnu.org/ml/gcc-patches/2006-05/msg00266.html, 2006.

[35] Karrenberg, R., and Hack, S. Whole-function vectorization. In *CGO* (2011), IEEE, pp. 141–150.

[36] Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. When polyhedral transformations meet simd code generation. *ACM SIGPLAN Notices 48*, 6 (2013), 127–138.

[37] Lapack. http://www.netlib.org/lapack/.

[38] Lattner, C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[39] Lim, A. W., and Lam, M. S. Maximizing parallelism and minimizing synchronization with affine partitioning. In *POPL'97* (1997).

[40] Llvm assembly language. http://llvm.org/docs/LangRef.html.

[41] Martinez Caamaño, J. M. *Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization*. Theses, Université de Strasbourg, Sept. 2016.

[42] Maslov, V. Delinearization: An efficient way to break multiloop dependence equations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1992), PLDI '92, ACM, pp. 152–161.

[43] Nuzman, D., and Zaks, A. Autovectorization in gcc–two years later. In *Proceedings of the 2006 GCC Developers Summit* (2006), Citeseer, pp. 145–158.

[44] Oancea, C. E., and Rauchwerger, L. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 509–520.

[45] Park, E., Cavazos, J., Pouchet, L.-N., Bastoul, C., Cohen, A., and Sadayappan, P. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming 41*, 5 (2013), 704–750.

[46] Pop, S., Cohen, A., and Silber, G.-A. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers*. Springer, 2005, pp. 218–232.

[47] Pouchet, L.-N. PoCC, the Polyhedral Compiler Collection, version 1.3, 2010. http://pocc.sourceforge.net.

[48] Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., and Vasilache, N. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices 46*, 1 (2011), 549–562.

[49] Pouchet, L.-N., and Yuki, T. PolyBench/C 4.1, 2015. http://polybench.sourceforge.net.

[50] Pugh, W. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 1991), Supercomputing '91, ACM, pp. 4–13.

[51] Pugh, W., and Wonnacott, D. Non-linear array dependence analysis. In *Languages, Compilers and Run-Time Systems for Scalable Computers*, B. Szymanski and B. Sinharoy, Eds. Springer US, 1996, pp. 1–14.

[52] Ramanujam, J., and Sadayappan, P. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing 16*, 2 (1992), 108–120.

[53] Rus, S., Rauchwerger, L., and Hoeflinger, J. Hybrid analysis: Static and dynamic memory reference analysis. In *ICS* (2002), IEEE Computer Society, pp. 251–283.

[54] Schweighofer, M. An algorithmic approach to schmüdgen's positivstellensatz. *Journal of Pure and Applied Algebra 166*, 3 (2002), 307 – 319.

[55] Sukumaran-Rajam, A. *Beyond the Realm of the Polyhedral Model: Combining Speculative Program Parallelization with Polyhedral Compilation*. Theses, Université de Strasbourg, Nov. 2015.

[56] Suriana, P. A. Fourier-motzkin with non-linear symbolic constant coefficients. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2016.

[57] Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J., and Upadrasta, R. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)* (2010).

[58] Venkat, A., Shantharam, M., Hall, M., and Strout, M. M. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, pp. 185:185–185:194.

[59] Wolfe, M. *High Performance Compilers for Parallel Computing*, 1st ed. Adison-Wesley, 1996.

[60] Xue, J. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.