Optimistic Delinearization of Parametrically Sized Arrays

Tobias Grosser
INRIA/ETH Zurich[*]
tobias.grosser@inf.ethz.ch

J. Ramanujam
Louisiana State University
ram@cct.lsu.edu

Louis-Noël Pouchet
The Ohio State University
pouchet@osu.edu

P. Sadayappan
The Ohio State University
sadayappan.1@osu.edu

Sebastian Pop
Samsung Austin R&D Center
sebpop@gmail.com

ABSTRACT

A number of legacy codes make use of linearized array references (i.e., references to one-dimensional arrays) to encode accesses to multi-dimensional arrays. This is also true of a number of optimized libraries and the well-known LLVM intermediate representation, which linearize array accesses. In many cases, the only information available is an array base pointer and a single dimensional offset. For problems with parametric array extents, this offset is usually a multivariate polynomial. Compiler analyses such as data dependence analysis are impeded because the standard formulations with integer linear programming (ILP) solvers cannot be used. In this paper, we present an approach to delinearization, i.e., recovering the multi-dimensional nature of accesses to arrays of parametric size. In case of insufficient static information, the developed algorithm produces run-time conditions to validate the recovered multi-dimensional form. The obtained access description enhances the precision of data dependence analysis. Experimental evaluation in the context of the LLVM/Polly system using a number of benchmarks reveals significant performance benefits due to increased precision of dependence analysis and enhanced optimization opportunities that are exploited by the compiler after delinearization.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processor – Compilers

Keywords

polyhedral analysis, linear memory layout, multi-dimensional arrays

1. INTRODUCTION

Dense multi-dimensional arrays are data structures widely used in programs, such as in scientific, engineering and image processing applications. Multi-dimensional arrays are native constructs in languages such as Java, FORTRAN or C99, but unfortunately not in languages like C++ or C89. Given a dense C99 multi-dimensional array `A[N][N]`, it can be equivalently viewed as a one-dimensional array `Alin[N*N]`: the memory layout of data described by a dense multi-dimensional array corresponds to a linear, consecutive chunk of data that can be viewed as a vector of memory locations. Therefore a particular memory cell reference in the multi-dimensional view, e.g., `A[i][j]` is *linearized* to `Alin[i*N + j]`.

Using a multi-dimensional view usually exposes simpler expressions for the subscript in each dimension (e.g., `i` and `j`) while the linearized view can complicate numerous compiler analyses and optimizations: `i*N + j` is a polynomial. Dependence and dataflow analysis on affine array accesses is a well known compiler technique [3], whereas performing the same analyses on polynomial expressions is significantly more complex and commonly not performed by many compilers. Therefore the ability to recover multi-dimensional array views is important to obtain precise data dependences. Such information enables a wide range of loop optimizations, and is essential for automatic parallelization, including GPGPU off-loading and MPI parallelization. For the latter two, the knowledge of multi-dimensional array views also eases precise memory footprint analyses, as well as the generation of communication code.

With languages not offering native support for multi-dimensional arrays, programmers must use explicitly linearized array index expressions with 1D arrays to encode algorithms that are naturally expressed using multi-dimensional arrays. Even with languages like C99 that provide native support for multi-dimensional arrays, the internal representation of compilers like Clang/LLVM only provides a linearized view of multi-dimensional arrays, for consistency across multiple input languages. Various legacy codes and several modern C++ template libraries also make use of linearized array references to encode accesses to multi-dimensional arrays. Listing 1 shows a function that initializes a sub-array of size $s_0 \times s_1 \times s_2$ located at offset $o_0 \times o_1 \times o_2$ inside a larger array of size $n_0 \times n_1 \times n_2$, implemented with C99 variable length arrays. Looking at the internal LLVM [5] representation after compiling with Clang (Listing 2), it may be seen that all array references have been linearized. LLVM's scalar evolution analysis [10] derives the access to `A` as $A[(n_2(n_1o_0 + o_1) + o_2) + n_1n_2i + jn_2 + k]$. Neither the array dimensionality nor the extent of individual dimensions is preserved.

We use a simple example to illustrate the impact of multi-dimensional array views on compiler optimization. The kernels in Listing 3 copy data from the odd rows of a 2D array to the even rows. The code has been deliberately made inefficient by accessing the data column-wise instead of row-wise. If a compiler can successfully prove the absence of data-dependences in this loop nest, it can interchange the two loops to avoid the inefficient data accesses and obtain significantly better performance. We compile three versions of the code. The first two versions mimic a multi-dimensional

[*]Finalized at the Dep. of Computer Science, ETH Zurich

ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
ACM 978-1-4503-3559-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2751205.2751248.

```
void set_subarray(unsigned n1, unsigned n2,
    unsigned o0, unsigned o1, unsigned o2,
    unsigned s0, unsigned s1, unsigned s2,
    float A[][n1][n2]) {
  for (unsigned i = 0; i < s0; i++)
    for (unsigned j = 0; j < s1; j++)
      for (unsigned k = 0; k < s2; k++)
S:        A[i + o0][j + o1][k + o2] = 1;
}
```

Listing 1: Initialization of sub-array (multi-dimensional)

```
define void @set_subarray(i32 %n1, i32 %n2, i32 %o0, i32 %o1,
  i32 %o2, i32 %s0, i32 %s1, i32 %s2, float* %A) {
  ; for i:
  ;   for j:
  ;     for k:
          %add = add i32 %k, %o2
          %add1 = add i32 %j, %o1
          %add2 = add i32 %i, %o0
          %0 = mul nuw i32 %n1, %n2
          %1 = mul nsw i32 %add2, %0
          %2 = mul nsw i32 %add1, %n2
          %idx1 = add i32 %1, %2
          %idx2 = add i32 %idx1, %add
          %idx3 = getelementptr float, float* %A, i32 %idx2
          store float 1.000000e+00, float* %idx3
  ;     endfor k
  ;   endfor j
  ; endfor i
}
```

Listing 2: Internal representation in Clang/LLVM

array using a macro that yields a linearized access into a single-dimensional array. Version #1 fixes the number of elements in the inner array dimension at compile time ($N = 20000$) whereas #2 uses a symbolic value for the size of the inner array dimension. Version #3 uses C99 variable-length arrays to make the two-dimensional symbolically sized shape explicit to the compiler. If compiled with the Intel icc 15.0 compiler both #1 and #3 are executed in 2 seconds, whereas #2 takes 15 seconds. icc is not able to perform the beneficial loop interchange for #2, but performs it for the other two versions. As loop interchange is clearly beneficial here, icc's inability to perform loop interchange is likely caused by imprecise and overly conservative data dependence information.

```
#define A_2D(o0, o1) A[(o0) * N + (o1)]
void oddEvenCopyLinearized(long N, float *A) {
#ifdef FIXED_SIZE
  N = 20000;
#endif
  for (long i = 0; i < N; i++)
    for (long j = 0; j < N; j++)
      A_2D(2 * j, i) = A_2D(2 * j + 1, i);
}
void oddEvenCopy2DArray(long N, float A[][N]) {
  for (long i = 0; i < N; i++)
    for (long j = 0; j < N; j++)
      A[2 * j][i] = A[2 * j + 1][i];
}
```

Listing 3: Dependence free odd-even copy within an array

In this paper, we address the *delinearization* problem: to infer equivalent multi-dimensional views of linearized arrays. This problem has been previously addressed by the work of Maslov [7] and Cierniak [2], but in a much more restricted context than we do. These previous efforts developed static analysis approaches that guaranteed a compile time solution under restricted scenarios: either produce a delinearized version, or declare infeasibility. In contrast, we significantly broaden the scope of programs that are delinearized, by taking an hybrid static/dynamic approach. Our analysis approach *optimistically* deduces a delinearized form, along with a set of validity conditions to be dynamically checked at runtime. If the runtime check passes, the deduced delinearized form is guaranteed to be equivalent to the linearized input, but if the dynamic check fails, no delinearized form is available. Although the check is dynamic, such an approach allows for conditional code with static program transformations applied to the delinearized form: the runtime check determines whether the statically optimized version using the delinearized form is executed or the unmodified version using the original linearized form is executed. We make the following contributions:

- We develop an effective approach to delinearization, inferring multi-dimensional views of arrays of parametric size; this approach also handles cases where statically proving the correctness of the recovered multi-dimensional view is infeasible.
- We implement and incorporate important parts of the approach into the Polly/LLVM system, thereby increasing the precision of data dependence analysis in Polly.
- We present experimental results demonstrating the benefit of the delinearization approach in enhancing performance optimization.

2. PROBLEM STATEMENT

Given a set of single dimensional array accesses with index expressions that are multivariate polynomials in terms of loop iterators and symbolic program parameters, and a set of corresponding iteration domains, derive an equivalent multi-dimensional view with linear array index expressions.

The view consists of 1) a multi-dimensional array definition (including the number of array dimensions and sizes for all but the outermost dimension), 2) for each original array access expression, a corresponding multi-dimensional array index expression. We impose the following set of conditions on the derived multi-dimensional view:

- (**R1**) **Affine:** The new access functions are affine in loop parameters and program parameters.
- (**R2**) **Equivalence:** For each array access, the memory location referenced by the original linearized subscript expression and the memory location obtained from the multi-dimensional view, after lowering it using the derived array sizes for a row-major array layout, are identical for all loop instances within the iteration space.
- (**R3**) **Within bounds:** The array subscript expressions (for all but the outermost dimension) are within the bounds of the multi-dimensional array, for all iteration instances within the iteration space.

For cases where the multi-dimensional view cannot be statically proven to be equivalent to the input linearized form, we derive a multi-dimensional view and provide a set of conditions under which the view is valid. $R1$ ensures that we can represent the resulting access expressions as affine integer maps. $R2$ ensures that the multi-dimensional form of the array has the same access characteristics as the single dimensional array. $R3$ ensures together with $R2$ that if we define a relation R between the elements of the linearized and the multi-dimensional view of the linearized array such that two elements are related if and only if they map to the same data location, this relation is always bijective. This property is important as it ensures that for each actual memory location there is only a single data location in our model; this is necessary for the correct computation of data dependences.

3. ARRAYS OF PARAMETRIC SIZE

We now present an algorithm to delinearize a 1D array reference in the form of a multivariate polynomial indexing expression into a set of affine indexing functions referencing a multi-dimensional array of shape $A[P_0][P_1]\dots[P_{n-1}]$, $P_i \in \mathcal{P}$ with $\mathcal{P}$ referring to the set of program parameters. We obtain a multi-dimensional array shape with the size of each dimension being defined by a single parameter, with multiple dimensions possibly sharing the same parameter.

3.1 Basic algorithm

We propose a basic algorithm consisting of the following steps:

1. Collect possible array size parameters
2. Derive dimensionality and array size
3. Compute multi-dimensional access functions
4. Derive validity conditions

We first collect information about possible array size parameters, symbolic values unknown at compile time that correspond to the size of an array dimension in the multi-dimensional view. To do this, we expand the given polynomial expression into a sum of products. From this sum, we extract all terms that contain both a loop induction variable and (possibly multiple) parameters. Those terms are interesting as the presence of a term that multiplies a parameter with an induction variable makes the expression non-affine. However, in case a parameter P is an array size parameter, P may be removed from the index expressions during delinearization, so that the original expression is turned into an access with affine subscript expressions. Consequently, we guess that P defines the size of at least one array dimension.

As the second step, we derive the dimensionality and the size of the array. To do this we start from the terms obtained in the previous step and assume all of them form products. In case a term is not a product, we treat it as a product with just a single factor. We remove from each term all factors that are non-parametric. The resulting terms are sorted according to the number of contained factors and we check that the terms with fewer factors symbolically divide the larger terms. If this is true, we assume the results of these symbolic divisions are the sizes of the array dimensions.

As the third step, we extract the access functions of the individual dimensions. We start with the original polynomial expression and first divide it symbolically by the size of the elements accessed. The resulting expression is then divided by the assumed dimension sizes starting with the innermost size. The remainder is the access function of the innermost dimension and the quotient is divided again by the size of the next array dimension. The new remainder is the access function of the second array dimension and the quotient is divided further. If no more dimension sizes are available, the last quotient becomes the access function of the outermost dimension.

As a last step, we derive the validity conditions. Up to this step, the delinearization we propose is an educated guess. It is only valid if $\forall i \in [1, n-1] : 0 \leq f_i(\mathbf{i}) < d_i$ holds, with n being the number of array dimensions computed, d_i being the size of dimension i and $f_i(\mathbf{i})$ being the access function of dimension i, which given a vector of loop induction variables and program parameters derives a subscript expression. To check if these conditions hold, we can simplify them taking into account the range of the surrounding loop induction variables. In simple cases this simplification yields $\top$, which means the delinearization has been statically proven to be unconditionally correct. In cases where this is not enough, the remaining conditions need to be emitted as run-time checks.

We illustrate the delinearization algorithm using the sub-array initialization example presented in the introduction, but now start from its single-dimension version (Listing 4). To recover the multi-dimensional nature of the access in statement S, we first expand

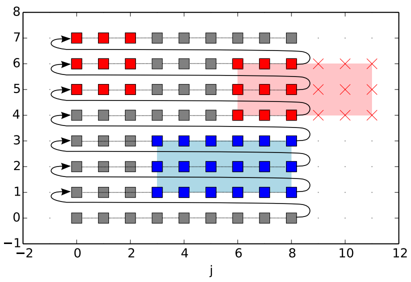


Figure 1: Sub-array accesses for different parameter values

the offset expression $(n_2(n_1o_0 + o_1) + o_2) + n_1n_2i + n_2j + k$ to a sum of products $n_2n_1o_0 + n_2o_1 + o_2 + n_1n_2i + n_2j + k$. Next, all products that involve induction variables are extracted, induction variables are removed and the products are sorted by the number of factors. This yields the set $\{n_1n_2, n_2\}$. As the smaller terms in this set evenly divide the larger ones, we assume a multi-dimensional array of shape `A[][n1][n2]`. We now use the new array shape to derive the individual index expressions. We do this by symbolically dividing the original offset expression by the size of the individual array dimensions, starting from the innermost dimension. As a first step we divide by n_2, which leaves us with a remainder $o_2 + k$, the index expression we assume for the innermost dimension. The quotient of the division is $n_1o_0 + o_1 + n_1i + j$. This quotient is now divided by n_1. The resulting remainder $o_1 + j$ allows us to derive $A[?][j + o_1][?]$ and the resulting quotient $o_0 + i$ allows us to derive $A[o_0 + i][?][?]$. The reconstructed full array access is $A[i + o_0][j + o_1][k + o_2]$.

```
void set_subarray(unsigned n1, unsigned n2,
    unsigned o0, unsigned o1, unsigned o2,
    unsigned s0, unsigned s1, unsigned s2,
    float A[]) {
  for (unsigned i = 0; i < s0; i++)
    for (unsigned j = 0; j < s1; j++)
      for (unsigned k = 0; k < s2; k++)
S:      A[(n2 * (n1 * o0 + o1) + o2)
          + n1 * n2 * i + n2 * j + k] = 1;
}
```

Listing 4: Initialization of sub-array (single-dimensional)

As a last step, we check the correctness of the delinearization by forming the following validity condition:

$$\forall i, j, k : \; 0 \leq i < s_0 \wedge 0 \leq j < s_1 \wedge 0 \leq k < s_2 : \\ 0 \leq k + o_2 < n_2 \wedge 0 \leq j + o_1 < n_1 \wedge 0 \leq i + o_0$$

Using isl [13], we simplify this condition to $o_1 \leq n_1 - s_1 \wedge o_2 \leq n_2 - s_2$, exploiting the fact that all parameters are given as unsigned types. As further simplifications are not possible at compile time, the remaining conditions need to be verified at run-time.

Figure 1 illustrates a two dimensional version of this example highlighting two sets of parameter values, one that satisfies the validity condition and one that does not. Both examples work on a 2D data array `A[n0][n1]` with $n_0 = 8 \wedge n_1 = 9$. The first set of parameter values is $o_0 = 1 \wedge o_1 = 3 \wedge s_0 = 3 \wedge s_1 = 6$, which yields $3 \leq 9 - 6$ and evaluates to $\top$. The corresponding set of data elements (illustrated in blue) are all within the bounds of the 2D array. However, of the accesses that correspond to the parameter values $o_0 = 4 \wedge o_1 = 6 \wedge s_0 = 3 \wedge s_1 = 6$ (red square) only the left half is within the array bounds. The right half accesses are out-of-bounds. In this case, the out-of-bounds accesses reference memory locations that correspond to the array elements $\{A[i, j] : 5 \leq i \leq 7 \wedge 0 \leq j \leq 2\}$ (red squares). This is problematic, as e.g., the data stored to `A[6][9]` affects the values read from `A[7][0]`. This relation is not visible in the delinearized program, which means the corresponding data dependences are not modeled and certain program transformations may be performed incorrectly. When checking the validity conditions we see that $o_1 \le n_1 - s_1 \Rightarrow 6 \le 9 - 6 \Rightarrow \bot$, which correctly shows that for this set of parameters we cannot rely on the delinearization.

3.2 Multiple array references

In case the kernel we analyze contains more than one access to the same array (e.g., identified by its base pointer), it is important to ensure that all accesses are properly delinearized using the same assumed array shape. Ensuring this requires only a slight adjustment of the algorithm. In the case of multiple arrays, we extract the terms from all arrays and derive the assumed array size from the combined terms. Using this common array size, we can once again derive the array accesses individually. The validity conditions are also derived individually, but redundant conditions are removed in a subsequent step. In case data accesses reference different arrays, we group the data accesses by the different arrays they access and analyze each group individually assuming the absence of aliasing between accesses to different arrays. To generate the run-time conditions we merge the constraints from the individual arrays, remove redundant constraints and generate a single run-time check to verify the analysis.

3.3 Subscripts containing size parameters

```
float A[][N][M];
for (i = 0; i < L; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < M; k++)
S1:   A[i][j][k] = ...;         // A[i N M + j M + k]

S2: A[1][1][1] = ...;           // A[N M + M + 1]
S3: A[0][0][M - 1] = ...;       // A[M - 1]
S4: A[0][N - 1][0] = ...;       // A[N M - M]
S5: A[0][N - 1][M - 1] = ...;   // A[N * M - 1]
```

Listing 5: Array sizes in subscripts (multi-dimensional)

Listing 5 shows an example of multi-dimensional array accesses where the subscript expressions contain array size parameters. In case these accesses are linearized (see comments) for an access `A[0][0][M-1]` with array size `A[][N][M]`, the algorithm presented in Section 3 derives the access `A[0][1][-1]`, as it associates multiples of `M` with the second dimension. This delinearization is invalid, as the subscript in the inner dimension becomes negative.

In general, there is always a set of delinearizations that differ in their derived subscript expressions, even though they all compute the same address expression. For the above example, the accesses `A[0][-1][2*M-1]`, `A[0][0][M-1]` and `A[0][1][-1]` all compute the same address expression. In fact, for a given two dimensional access and an arbitrary $k \in \mathbb{N}$ the following holds:

$$\begin{aligned}
& A[f_0][f_1] && \text{with } A[\,][s_1]\\
=\ & A[f_0 s_1 + f_1] = A[(f_0 - k)s_1 + (ks_1 + f_1)] && \text{with } A[\,]\\
=\ & A[f_0 - k][ks_1 + f_1] && \text{with } A[\,][s_1]
\end{aligned}$$

For d-dimensional accesses and any pair of neighboring dimensions $(t, t+1), t \in [0, d-2]$ the following equality holds for all $k_t \in \mathbb{N}$:

$$\&A[\ldots, f_t, f_{t+1}, \ldots] = \&A[\ldots, f_t - k_t, k_t s_{t+1} + f_{t+1}, \ldots]$$

This means there exists a set of values $k_t, t \in [0, d-2]$ which can be arbitrarily chosen to generate an infinite number of array accesses that all yield the same linearized address expression. However, for a specific set of loop iterators and parameters, not all k_t values ensure array accesses within bounds. We need to find the right values of k_t that avoid out-of-bounds accesses. One idea is to look at the loop bounds and to statically derive the right values of k_t. This is possible as long as the range of the subscript expression is known, but causes problems for accesses such as `A[N * i + N + p]`, which might be modeled as an array access `A[i + 1][p]` to an array of shape `A[][N]` in case $0 \le p < N$ holds, but which would better be modeled as `A[i][N + p]` in case $-N \le p < 0$ holds. In general it is not possible to derive an optimal value for k without knowledge about the values p can take. In some cases (e.g., $-10 \le p < N - 10$) there is not even a single optimal value for k. To still be able to model such cases we can create a piece-wise delinearization that chooses the correct value of k depending on the values of the subscript expressions. For the 2D case, we could be tempted to use a mapping $(f_0, f_1) \to (f_0 + k, -ks_1 + f_1) \mid \exists k : ks_1 \le f_1 < (k+1)s_1$, which models all possible values of k. Unfortunately, the product between k and s_1 is non-affine and consequently this map cannot be represented as an affine integer map. However, if we bound k such that $k \in [k_l, k_u]$ with k_l, k_u being known integer values, we can model this map with a finite number of affine pieces.

$$(f_0, f_1) \to \begin{cases}
(f_0 + k_l, -k_l s_1 + f_2) & f_1 < k_l s_1\\
\vdots & \\
(f_0 + (\text{-}1), \text{-}(\text{-}1)s_1 + f_2) & (\text{-}1)s_1 \le f_1 < 0\\
(f_0, f_1) & 0 \le f_1 < 1s_1\\
(f_0 + 1, \text{-}(1)s_1 + f_2) & 1s_1 \le f_1 < 2s_1\\
\vdots & \\
(f_0 + k_u, -k_u s_1 + f_2) & k_u s_1 \le f_1
\end{cases}$$

For d-dimensional accesses, we now define a set of maps $M_t, t \in [0, d-2]$, where a map M_t is an identity map with dimension t and $t+1$ modified to use a generalized version of the above mapping. Each M_t approximates a map $(\ldots, f_t, f_{t+1}, \ldots) \to (\ldots, f_t + k, -k * s_{t+1} + f_{t+1}, \ldots) \mid \exists k : ks_{t+1} \le f_1 < (k+1)s_{t+1}$ using a finite set of affine pieces. Starting from the highest t, we apply all maps M_t one by one to the delinearized (i.e., the multi-dimensional view) accesses. Using the original algorithm on the example given in Listing 5, we obtain the following set of delinearized accesses: $\{S1(i,j,k) \to A(i,j,k),\ \ S2() \to A(1,1,1),\ \ S3() \to A(0,1,-1),\ S4() \to A(1,-1,0), S5() \to A(1,0,-1)\}$. After applying a set of maps M_t generated with values $k_{t,l} = 0, k_{t,u} = 0$ chosen to only cover two cases, one with no transformation and one with a single multiple of the problem size parameter added, we obtain the following delinearized accesses:

$$S1(i,j,k) \to \begin{cases}
A(i-1, N+j-1, M+k) & k \le -1 \wedge j-1 \le -1\\
A(i, j-1, M+k) & k \le -1 \wedge j-1 \ge 0\\
A(i-1, N+j, k) & k \ge 0 \wedge j \le -1\\
A(i,j,k) & k \ge 0 \wedge j \ge 0
\end{cases}$$

$$\begin{aligned}
S2() &\to A(1,1,1), & S3() &\to A(0,0,M-1)\\
S4() &\to A(0,N-1,0), & S5() &\to A(0,N-1,M-1)
\end{aligned}$$

S2, S3, S4 and S5 show directly the correct delinearization. The access function for S1 is now slightly more complicated, but the three additional cases only apply under conditions that are removed when simplifying the access under the constraints implied by the iteration domain of S1. After these simplifications we obtain the mapping $S1(i,j,k) \to A(i,j,k)$ for $S1$. So the piecewise mappings have all been statically reduced to maps with just a single piece.

3.4 Arrays of size $A[\,][\beta_1 P_1][\beta_2 P_2]$

In certain cases (e.g., resizing of images) we may have array sizes of the form $A[\,][\beta_1 P_1][\beta_2 P_2]$, $P_i \in \mathcal{P}$, $\beta_i \in \mathbb{N}$. Accesses to such arrays would be delinearized to an access $A[\beta_1 f_0][\beta_2 f_1][f_2]$, into an array of size $A[\,][P_1][P_2]$. As f_1 can be in the range $0 \leq f_1 < \beta_1 P_1$, the expression $\beta_2 f_1$ may not fit into the new range. To address this, we can find the *gcd* of the values in each dimension and use it to adjust the array sizes. Specifically, if all subscript expressions in a certain dimension can be divided by a value x, we can divide all of them by x and multiply the size of the next innermost dimension by x. This transformation is always beneficial in the sense that it only increases the chance that the delinearization will be correct. As it reduces the range of the subscript expression, the subscript expression is more likely to fit into the ranges implied by the array size. Similarly, as we increase the size of the inner dimension the corresponding subscript expressions on this dimension are also more likely to fit in.

4. PARAMETER + CONSTANT

We look now at a specific case, where the shape of the array is of the form $A[P_0 + \alpha_0] \dots [P_{n-1} + \alpha_{n-1}]$ with $\forall i \in [0, n-1] : P_i \in \mathcal{P}$, $\alpha_i \in \mathbb{N}$, with P_i being different for different values of i.[1] As an example we show in Listing 6 a simplified 3D stencil computation which computes the average over the elements in a diagonal stencil and which uses a one element border around the actual data elements to avoid the need for special boundary statements.

```
int In[Q+2][R+2][S+2]; int Out[Q+2][R+2][S+2];

for (int i = 1; i <= Q; i++)
  for (int i = 1; i <= R; i++)
    for (int i = 1; i <= S; i++)
      Out[i][j][k] = 0.33f * (In[i][j][k]
        + In[i+1][j+1][k+1] + In[i-1][j-1][k-1]);
```

Listing 6: Dimensions of size $P_i + \alpha_i$ (multi-dimensional)

```
int In[]; int Out[];

for (int i = 1; i <= Q; i++)
  for (int i = 1; i <= R; i++)
    for (int i = 1; i <= S; i++)
      Out[i*R*S+2*i*S+2*i*R+4*i+j*S+j*2+k] =
        0.33f * (In[i*R*S+2*i*S+2*i*R+4*i+j*S+j*2+k]
                + In[7+2*j+k+2*R+3*S+j*S+
                    R*S+4*i+2*R*i+2*S*i+R*S*i]
                + In[-7+2*j+k-2*R-3*S+j*S-R*S
                    +4*i+2*R*i+2*S*i+R*S*i]);
```

Listing 7: Dimensions of size $P_i + \alpha_i$ (linearized)

When analyzing the linearized access to `Out`, as visible in Listing 7, two problems become visible. First of all, the previous algorithm fails to guess an array size, as the terms R, S and RS all appear in products that contain induction variables. Our previous approach can consequently not define an order on the parameters that allows it to assign parameters to array dimensions. Assuming we could still derive an array shape (e.g., `Out[][R][S]`), we obtain from the remaining algorithm the access `Out[i][2i + j][4 i + 2 j + k + 2 i R]`. This delinearization is incorrect. As it has been derived according to the wrong array size, it causes out-of-bound accesses and even fails to fully eliminate non-affine terms in the subscript expressions. Before presenting the general approach to delinearize polynomial expressions to d-dimensional array shapes of the form $A[P_0 + \alpha_0] \dots [P_{d-1} + \alpha_{d-1}]$, $P_i \in \mathcal{P}$, $\alpha \in \mathbb{N}$, we present the special case of two and three dimensions.

An access to a two dimensional array $A[f_0(\mathbf{i})][f_1(\mathbf{i})]$ with shape $A[\,][P_1 + \alpha_1]$ corresponds to the single dimensional array access $A[f_0(\mathbf{i})(P_1 + \alpha_1) + f_1(\mathbf{i})]$, which after expansion becomes the access $A[f_0(\mathbf{i})P_1 + f_0(\mathbf{i})\alpha_1 + f_1(\mathbf{i})]$. However, it is unlikely that this structure is preserved. The only structure that can be assumed is a sum of terms $g_{\{1\}}(\mathbf{i})P_1 + g_{\emptyset}(\mathbf{i})$ where each term contains a different subset of the program parameters. In the general case we write this expression as $\sum_{S \in \wp([0,f-1])} (g_S(\mathbf{i}) \prod_{s \in S} P_s)$, where f is the number of parameters, $\wp([0, f-1])$ is the powerset of $[0, f-1]$, the different $g_S(\mathbf{i})$ are expressions in loop induction variables, and the different P_s are program parameters. To delinearize this polynomial expression we need to recover expressions $f_0(\mathbf{i}), f_1(\mathbf{i}), \alpha_1$ as a function of g_x's. As $f_0(\mathbf{i})$ is the only coefficient to P_1, recovering the relation $f_0(\mathbf{i}) = g_{\{1\}}(\mathbf{i})$ is easy. The second equality we can obtain is $g_{\emptyset}(\mathbf{i}) = f_0(\mathbf{i})\alpha_1 + f_1(\mathbf{i})$. With $f_0(\mathbf{i})$ plugged in we obtain $g_{\emptyset}(\mathbf{i}) = g_{\{1\}}(\mathbf{i})\alpha_1 + f_1(\mathbf{i})$, which allows us to express $f_1(\mathbf{i})$ as a function of α_1: $f_1(\mathbf{i}) = g_{\emptyset}(\mathbf{i}) - g_{\{1\}}(\mathbf{i})\alpha_1$. For different values of α_1 we obtain different array sizes and the corresponding delinearizations, which all are lowered to the very same linearized function, perform the same memory accesses and consequently model the program behavior correctly. However, depending on the iteration space boundaries only certain delinearizations ensure the absence of out of bounds accesses. As boundary offsets are commonly small and there is only one value α_1 to verify, it is possible to scan a certain number of α_1 by either statically checking for valid delinearizations or possibly even by generating run-time versioned code for different values of α_1.

Looking at the three dimensional case, we observe that an access $A[f_0(\mathbf{i})][f_1(\mathbf{i})][f_2(\mathbf{i})]$ to an array of shape $A[\,][P_1 + \alpha_1][P_2 + \alpha_2]$ has, after linearization and expansion, the form:

$$f_0(\mathbf{i})P_1P_2 + f_0(\mathbf{i})P_1\alpha_2 + f_0(\mathbf{i})P_2\alpha_1 + f_0(\mathbf{i})\alpha_1\alpha_2 + \\ f_1(\mathbf{i})P_2 + f_1(\mathbf{i})\alpha_2 + f_2(\mathbf{i})$$

It corresponds to the polynomial expression:

$$g_{\{1,2\}}(\mathbf{i})P_1P_2 + g_{\{1\}}(\mathbf{i})P_1 + g_{\{2\}}(\mathbf{i})P_2 + g_{\emptyset}(\mathbf{i})$$

From the single term that contains P_1P_2, the product of all symbolic parameters defining the array sizes, we recover the relation $f_0(\mathbf{i}) = g_{\{1,2\}}(\mathbf{i})$. Assuming P_1 is the outermost parameter, we obtain the value of α_2 from the single term that contains P_1, but not P_2: $g_{\{1\}}(\mathbf{i}) = f_0(\mathbf{i})\alpha_2 \Rightarrow \alpha_2 = g_{\{1\}}(\mathbf{i})/f_0(\mathbf{i}) = g_{\{1\}}(\mathbf{i})/g_{\{1,2\}}(\mathbf{i})$. Looking at the P_2 terms, we obtain the relation $g_{\{2\}}(\mathbf{i}) = f_0(\mathbf{i})\alpha_1 + f_1(\mathbf{i})$. This allows us to derive $f_1(\mathbf{i}) = g_{\{2\}}(\mathbf{i}) - f_0(\mathbf{i})\alpha_1 = g_{\{2\}}(\mathbf{i}) - g_{\{1,2\}}(\mathbf{i})\alpha_1$. Again, an expression containing α_1 as a free variable. To obtain $f_2(\mathbf{i})$ we look at the terms without any parameters. Here we have $g_{\emptyset}(\mathbf{i}) = f_0(\mathbf{i})\alpha_1\alpha_2 + f_1(\mathbf{i})\alpha_2 + f_2(\mathbf{i})$ from which we can derive $f_2(\mathbf{i}) = g_{\emptyset}(\mathbf{i}) - f_0(\mathbf{i})\alpha_1\alpha_2 - f_1(\mathbf{i})\alpha_2 = g_{\emptyset}(\mathbf{i}) - f_0(\mathbf{i})\alpha_1\alpha_2 - (g_{\{2\}}(\mathbf{i}) - f_0(\mathbf{i})\alpha_1)\alpha_2 = g_{\emptyset}(\mathbf{i}) - f_0(\mathbf{i})\alpha_1\alpha_2 - g_{\{2\}}(\mathbf{i})\alpha_2 + f_0(\mathbf{i})\alpha_1\alpha_2 = g_{\emptyset}(\mathbf{i}) - g_{\{2\}}(\mathbf{i})\alpha_2$. As α_1 cancels out, we can unambiguously derive $f_2(\mathbf{i})$. We can conclude that delinearizing to a three-dimensional array shape does not introduce more freedom. Only α_1 remains unknown and different values may need to be explored.

We now present with Algorithm 1 a general algorithm to delinearize polynomial expressions to array shapes of arbitrary dimensionality. We first collect the set of possible array size parameters and then try for each order to find a valid delinearization. To check if a valid delinearization exists, we first compute $f_0(\mathbf{i})$ and use it to try to derive a set of consistent α values. If we succeed, we derive subscript expressions and run-time conditions. In case the run-time condition is not a contradiction, we assume we found a

[1]This does not include shapes such as `A[][N+1][N+1]`

Algorithm 1: Derive a delinearization

```
Data: A polynomial expression in function of induction
      variables and parameters, a list of array size parameters
Result: A set of values $\alpha_k, k \in [1, d-1]$, index expressions
        $f_k, k \in [0, d-1]$ and set of array size parameters
        $P_k, k \in [1, d-1]$ or an error if no delinearization found.
collect possible array sizes parameters;
foreach permutation of array sizes parameters do
    derive $f_0$;
    alpha = derive alpha values;
    if alpha $\neq$ [ ] then
        derive subscript expressions;
        derive run-time condition;
        if run-time condition is a contradiction then
            continue;
        else
            return subscript expressions, run-time-condition,
                   array-sizes
return No delinearization found!
```

valid delinearization and finish, otherwise we try the next permutation. To obtain the set of possible array size parameters, we take the expanded version of the polynomial expression and look again for parameters that are multiplied with a loop induction variable.

For the remaining analysis it is necessary to understand the shape of the analyzed polynomial expression. Specifically, we must be able to group them in a manner that each term is the product between a subset of the assumed array size parameters and an expression $g_?(\mathbf{i})$ in loop indexes, non array-size parameters and integer constants:

$$\begin{aligned} g_\emptyset(\mathbf{i}) + {} & g_{\{1\}}(\mathbf{i})P_1 + g_{\{2\}}(\mathbf{i})P_2 + \cdots + g_{\{d-1\}}(\mathbf{i})P_{d-1} \\ & + g_{\{1,2\}}(\mathbf{i})P_1P_2 + g_{\{1,3\}}(\mathbf{i})P_1P_3 + \cdots + g_{\{2,3\}}(\mathbf{i})P_2P_3 + \ldots \\ & + g_{[1,d-1]}(\mathbf{i})P_1 \ldots P_{d-1} \\ & = \sum_{K \in \mathcal{P}([1,d-1])} \Big(g_K(\mathbf{i}) \prod_{k \in K} P_k \Big) \end{aligned}$$

We now want to express the previous polynomial as a d-dimensional access $A[f_0(\mathbf{i})] \ldots [f_{d-1}(\mathbf{i})]$ to an array of size $A[\,][P_1 + \alpha_1] \ldots [P_{d-1} + \alpha_{d-1}]$. To do so, look at how such an array is linearized:

$$\begin{aligned} & f_0(\mathbf{i})(P_1 + \alpha_1)(P_2 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\ & + f_1(\mathbf{i})(P_2 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\ + \ldots\ & + f_{d-2}(\mathbf{i})(P_{d-1} + \alpha_{d-1}) \\ & + f_{d-1}(\mathbf{i}) \\ & = \sum_{j \in [0,d-1]} \Big(f_j(\mathbf{i}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \Big) \end{aligned}$$

and assume this linearized form yields the same access computation as the one-dimensional expression we want to delinearize:

$$\begin{aligned} \sum_{K \in \mathcal{P}([1,d-1])} \Big(g_K(\mathbf{i}) \prod_{k \in K} P_k \Big) &= \sum_{j \in [0,d-1]} \Big(f_j(\mathbf{i}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \Big) \\ &= \sum_{j \in [0,d-1]} \sum_{K \in \mathcal{P}([j+1,d-1])} \Big(f_j(\mathbf{i}) \prod_{k \in K} P_k \prod_{k \in [j+1,d-1] \setminus K} \alpha_k \Big) \end{aligned}$$

We now equate terms that contain the same set of parameters and, assuming the parameters to be positive, drop the common parame-

$$\begin{aligned} \frac{g_S(\mathbf{i})}{g_T(\mathbf{i})} &= \frac{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus S} \alpha_x \Big)}{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)} = \frac{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \alpha_k \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)}{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)} \\ &= \frac{\alpha_k \cdot \sum\limits_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)}{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)} = \frac{\alpha_k \cdot \sum\limits_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)}{\sum\limits_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod\limits_{x \in [j+1,d-1] \setminus T} \alpha_x \Big)} \\ &= \alpha_k \end{aligned}$$

Figure 2: Deriving α_k

ters on both sides. As a result, we obtain $\forall K \in \mathcal{P}([1, d-1])$:

$$g_K(\mathbf{i}) = \sum_{\substack{j \in [0,d-1] \\ \wedge K \subseteq [j+1,d-1]}} \Big(f_j(\mathbf{i}) \prod_{k \in [j+1,d-1] \setminus K} \alpha_k \Big)$$

Having established this set of equalities, we start to relate our terms $g_?(\mathbf{i})$ to the terms $f_?(\mathbf{i})$ and $\alpha_?$ that we want to derive. We first derive $f_0(\mathbf{i}) = g_{[1,d-1]}(\mathbf{i})$, which can be trivially derived by setting $K = [1, d-1]$ in the previous equation.

Algorithm 2: Derive alpha values

```
Data: A dimensionality $d$, a set of expressions $g_s(\mathbf{i})$
Result: A list of values $\alpha_k, k \in [2, d-1]$ or [] in case of
        inconsistencies
foreach $k \in [2, d-1]$ do
    if $g_{[1,d-1]}$ not evenly divides $g_{[1,d-1]\setminus\{k\}}(\mathbf{i})$ then
        return [];
    $\alpha_k = g_{[1,d-1]\setminus\{k\}}(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i})$;
    foreach $S \in \mathcal{P}([2, k-1] \setminus (\emptyset \cup ([1, d-1] \setminus \{k\}))$ do
        if $g_{[1,d-1]}(\mathbf{i})$ not evenly divides S then
            return [];
        $\alpha'_k = g_S(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i})$;
        if $\alpha'_k \neq \alpha_k$ then
            return [];
return $\{k \to \alpha_k : k \in [2, d-1]\}$
```

We derive the values α_k from the terms $g_{[1,d-1]\setminus\{k\}}(\mathbf{i})$. In the four-dimensional case such terms have the form:

$$\begin{aligned} & g_{\{2,3,4\}}(\mathbf{i}) = \alpha_1 f_0(\mathbf{i}) + f_1(\mathbf{i}) \\ & g_{\{1,3,4\}}(\mathbf{i}) = \alpha_2 f_0(\mathbf{i}) && \Rightarrow \alpha_2 = g_{\{1,3,4\}}(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i}) \\ & g_{\{1,2,4\}}(\mathbf{i}) = \alpha_3 f_0(\mathbf{i}) && \Rightarrow \alpha_3 = g_{\{1,2,4\}}(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i}) \\ & g_{\{1,2,3\}}(\mathbf{i}) = \alpha_4 f_0(\mathbf{i}) && \Rightarrow \alpha_4 = g_{\{1,2,3\}}(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i}) \end{aligned}$$

In general $\alpha_k, k \in [2, d-1]$ is $\alpha_k = g_{[1,d-1]\setminus\{k\}}(\mathbf{i}) / g_{[1,d-1]}(\mathbf{i})$. Similar to the two and three dimensional case, we cannot derive a value for α_1, as we do not know the value of $f_1(\mathbf{i})$. However, for higher dimensional cases we can make an interesting observation. The values of α_k can not just be obtained by the equalities presented above. In fact, there is a larger set of equalities that all need to return the same values α_k for an array view to be a valid delinearization. Specifically, to derive $\alpha_k, k \in [1, d-1]$ we can choose any pair of sets $(S, T), \emptyset \subset S \subseteq [1, k-1] \wedge T = S \cap \{k\}$ which can be used to compute α_k as shown in Figure 2. The following lists the closed form expressions that compute α_2 and α_3 for the 4D case:

$$\alpha_2 = g_{\{1,3\}}(\mathbf{i})/g_{\{1,2,3\}}(\mathbf{i}), \qquad \alpha_3 = g_{\{1\}}(\mathbf{i})/g_{\{1,3\}}(\mathbf{i})$$
$$\alpha_3 = g_{\{2\}}(\mathbf{i})/g_{\{2,3\}}(\mathbf{i}), \qquad \alpha_3 = g_{\{1,2\}}(\mathbf{i})/g_{\{1,2,3\}}(\mathbf{i})$$

We can see that α_3 can be derived in multiple ways. The delinearization is correct only if all yield the same result. By checking all of them and comparing them, we can validate the order of the array size parameters. Algorithm 2 gives the full algorithm used to obtain the different alpha values.

After deriving the different values of α_k, we can now derive the terms $f_j, j \in [2, d-1]$ by looking at the terms $g_{[j+1,d-1]}(\mathbf{i})$ (only interesting terms listed):

$$\begin{aligned}
g_{\{1,\ldots,d-1\}}(\mathbf{i}) &= f_0(\mathbf{i}) \\
g_{\{2,\ldots,d-1\}}(\mathbf{i}) &= \alpha_1 f_0(\mathbf{i}) + f_1(\mathbf{i}) \\
g_{\{3,\ldots,d-1\}}(\mathbf{i}) &= \alpha_1\alpha_2 f_0(\mathbf{i}) + \alpha_2 f_1(\mathbf{i}) + f_2(\mathbf{i}) \\
&= \alpha_2 g_{\{2,\ldots,d-1\}}(\mathbf{i}) + f_2(\mathbf{i}) \\
g_{\{j,\ldots,d-1\}}(\mathbf{i}) &= \alpha_{j-1} g_{\{j-1(\mathbf{i}),\ldots,d-1\}}(\mathbf{i}) + f_{j-1}(\mathbf{i}) \\
g_{\emptyset}(\mathbf{i}) &= \alpha_{d-1} g_{\{d-1\}}(\mathbf{i}) + f_{d-1}(\mathbf{i})
\end{aligned}$$

from which we derive $f_j(\mathbf{i}) = g_{[j+1,d-1]}(\mathbf{i}) - \alpha_j g_{[j,d-1]}(\mathbf{i})$. The general algorithm (Algorithm 3) is straightforward, as it mainly uses the equalities just given to derive the relevant values. As a last step, we obtain the set of necessary run-time conditions. This step is unchanged from Section 3.

Algorithm 3: Derive subscript expressions

```
Data: A dimensionality $d$, a set of expressions
    $g_s(\mathbf{i}), s \in \mathcal{P}([0, d-1])$, a set of values $\alpha_k, k \in [2, d-1]$
Result: A set of expressions $f_k(\mathbf{i}), k \in [0, d-1]$
$f_0(\mathbf{i}) = g_{[1,d-1]}(\mathbf{i})$;
/* The next line assumes $\alpha_1 = 0$.                    */
$f_1(\mathbf{i}) = g_{2,d-1]}(\mathbf{i})$;
foreach $j \in [2, d-1]$ do
    $f_j(\mathbf{i}) = g_{[j+1,d-1]}(\mathbf{i}) - \alpha_j g_{[j,d-1]}(\mathbf{i})$
return $\{j \to f_j(\mathbf{i}) : j \in [0, d-1]\}$
```

5. EVALUATION

Essential parts of the presented approach have been implemented within LLVM and Polly [4]. In our implementation, LLVM's scalar evolution analysis has been used to perform the necessary transformations of index expressions, for example to extract the array size parameter candidates or to perform the division and remainder computations for obtaining subscript expressions. Besides basic delinearization support (Section 3.1), we also implemented support for array size parameters in the index expressions (Section 3.3), as well as support for deriving a unique array shape for a set of accesses (Section 3.2). We also have full support for the generation of run-time conditions that validate the delinearization. For the generation of run-time conditions, we rely on a new AST generator developed as part of isl [13] and use its support to generate AST expressions from user-provided integer sets. This feature allows us to use isl to compute the set of run-time constraints that need to be checked, the AST generator to generate optimal code for them, and Polly's code generation back-end to translate the resulting AST expressions to LLVM IR. One optimization that has shown to be useful for reducing the complexity of run-time conditions is to use isl to remove constraints that are only valid for parameter values for which no memory access is executed. This is obviously valid. In case no data access is executed, we cannot possibly model this access incorrectly. Finally, even though our implementation is reasonable robust, the delinearization problem itself is not always nicely exposed by LLVM. For example, sometimes LLVM can not prove that certain symbols are loop invariant, the use of different integer types causes type casts in subscript expressions or information about the absence of integer wrapping is not available. The manual modifications required to expose the delinearization problems will be stated for each experiment. We generally assume the absence of integer wrapping.

5.1 C99 arrays in PolyBench/C

We tested the implementation of the delinearization on all 30 PolyBench/C 3.2 kernels [11] with the use of C99 variable length arrays enabled (`-DPOLYBENCH_USE_C99_PROTO`). From the 30 kernels, Polly could correctly recover (after removal of ternary conditions and the use of 'long' to avoid spurious casts), the multidimensional view of arrays in 28 of them (Figure 3b). Two kernels (ludcmp, fdtd-apml) are currently skipped, due to the array size itself being of the form $N+1$. However, with both of these kernels, using either two dimensional arrays or arrays with different parameters in each size declaration, the approach proposed in Section 4 is applicable and would allow us to extend Polly to handle these cases as well. It is also interesting to note that the PolyBench/C code is written in a way that almost all delinearizations are statically provable. Nevertheless, our delinearization concluded that run-time checks are necessary for six benchmarks (correlation, covariance, 2mm, doitgen, symm, and reg_detect). On looking closer as to why run-time checks are still generated, we understand that for the first five benchmarks in the original PolyBench/C source code certain parameters have been accidentally swapped in the array declarations and loop bounds. This unintentionally changed the semantics of the loop kernels in a way that only if a certain relation between the different parameter holds (e.g., the matrices are square), the execution does not inhibit out-of-bound accesses. The run-time conditions computed directly reflect those conditions and ensure that only in such cases the optimized loop is used. For the last benchmark, reg_detect, a condition $length > 0$ is derived which is required as `S2` will access negative locations in the innermost dimension in case `length` is smaller or equal to zero. This is visible in Listing 8. Even though not foreseen, these examples show the benefits of the approach to delinearization. Not only did it prevent a possible mis-compilation, but it also ensured that we could still optimize it even though there exists a set of parameter values under which this optimization is not correct.

```
    for (j = 0; j <= n - 1; j++)
      for (i = j; i <= n - 1; i++) {
S0:       sum_diff[j][i][0] = ...
          for (cnt = 1; cnt <= length - 1; cnt++)
S1:             sum_diff[j][i][cnt] = ...
S2:         ... = sum_diff[j][i][length - 1];
      }
```

Listing 8: PolyBench/C's reg_detect requires length > 0.

To understand if the more precise access information enables additional compiler transformations we optimized the PolyBench/C test suite with 'clang -O3', 'clang -O3 -polly', and 'clang -O3 -polly -delinearize' and run the resulting binaries single-threaded on an Intel Xeon E5430 CPU. The results in Figure 3a show that delinearization has a very visible impact on performance. Several benchmarks show speedups of 5.0x and beyond, but we also see for trmm and floyd-warshall significant performance reductions due to additional loop transformations that became legal, but that have been poorly chosen by Polly. ludcmp and fdtd-apml, the benchmarks where we do not yet delinearize the kernels, do not show any performance impact. Also, benchmarks with run time checks, e.g., covariance and 2mm, show visible performance changes, which means their run-time check is evaluated to true. There is also a set of kernels that do not see performance changes. In general this is because Polly's current set of optimizations do not affect any of these kernels.

5.2 C++ template libraries – boost::ublas

We also evaluated our approach in the context of C++ template libraries. C++ template libraries are widely used to raise the level of abstraction in programming, but at the same time they make it a lot harder to gather information about the execution of a program via analysis of the source code. In general, heavy inlining is required to remove C++ iterators, virtual method calls as well as other tools of abstraction. Only after these have been removed can a compiler possibly understand the memory access pattern of a program. However, at this point of the compilation, high level information about the original array shapes is not available any more.

For our evaluation we have chosen boost::ublas, a C++ template library for dense, packed, and sparse matrices that uses efficient code generation via expression templates to avoid unnecessary memory and object allocation. For our evaluation, we implemented a simple dgemm kernel in boost::ublas and compiled it in production mode `-DNDEBUG -DBOOST_UBLAS_NDEBUG` and without exceptions to evaluate how different compilers can optimize this code. We considered gcc 4.8.3, icc 15.0 and clang+Polly pre-3.6 (r226126) and run our experiment on an Intel i7-3520M. For Polly we ensured the inliner is run *before* Polly is executed and we manually performed some simple manual LICM to expose the delinearization problem. As visible in Figure 3c, the binaries produced by the different compilers all run in 2.2 seconds. With Polly and our delinearization approach enabled, we achieve a speedup of 1.8x, just by applying some simple cache tiling.

5.3 Julia

We also considered Julia [1], a dynamic high-level language for scientific computing. One specialty of Julia is that it is very easy to write generic code that can be specialized for different data types. We used this feature to evaluate our approach on different data types, again using a gemm kernel as the computational pattern. As with the ublas example, after some simple loop-invariant code motion is performed manually, delinearizing the array accesses yields the expected results, run-time checks are emitted (and required) and the Julia kernel can be optimized with Polly. For matrices of size 1024×1024, Polly's default optimizations (mainly loop tiling) already give speedups from 3.5x–5.0x across all different data types, compared to compilation without delinearization, where Polly is not able to apply any transformation. Even though Julia's performance is still far from a tuned library such as ATLAS, reaching reasonable performance also for data-types that commonly lack specialized blas libraries is valuable. As arrays in Julia are commonly of parametric size, understanding their structure at the IR level is essential to enable a wide range of loop optimizations.

5.4 UTDSP

The UTDSP [6] benchmark evaluates the ability of C compilers to generate efficient code for kernels and applications from the digital signal processing domain. UTDSP provides each kernel in multiple versions, including versions that are specialized for a specific input size by relying on fixed-size multi-dimensional arrays and versions relying on pointer arithmetic, which could conceptually support inputs of varying size. To evaluate the delinearization algorithm, we modified the UTDSP edge detection benchmark to take problem sizes as command line arguments and use these sizes to both dynamically allocate the data arrays and to provide parametric bounds for the computation. In addition, we also replaced a scalar summation variable with a direct summation into the output image to remove spurious scalar dependences that can complicate later optimizations. We also ensured consistent use of 64-bit integer types to avoid complications due to sign-extensions and truncations.

Listing 9 shows the core loop of the UTDSP edge detection kernel used for our evaluation. The loop implements a generic convolution, that is executed three times. First, a Gaussian convolution is performed to smoothen the image; then, both a horizontal and a vertical Sobel filter are applied. When analyzing the kernel with Polly, three interesting compute statements are detected. They have been marked as `S0`, `S1` and `S2`. When recovering the data accesses in these statements without using our delinearization approach, the repeating addition of `NN` into the pointer variables yields access functions that contain products between the parameter `NN` and (virtual) loop induction variables. Such non-affine functions prevent precise dependence analysis and parallelism detection.

Using our delinearization approach, we recognize the variables `output_image`, `input_image` and `kernel` as base pointers of two dimensional array accesses. This allows us to derive the following, possibly multi-dimensional, but always affine access functions:

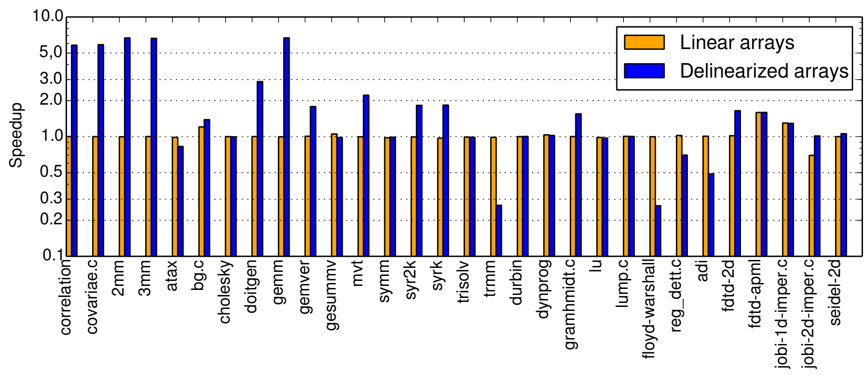
$$\begin{aligned}
S_0(r,c) &\rightarrow_{\text{write}} \text{output\_image}(\text{dead\_rows}+r, \text{dead\_cols}+c)\\
S_1(r,c,i,j) &\rightarrow_{\text{dead}} \text{output\_image}(\text{dead\_rows}+r, \text{dead\_cols}+c)\\
S_1(r,c,i,j) &\rightarrow_{\text{write}} \text{output\_image}(\text{dead\_rows}+r, \text{dead\_cols}+c)\\
S_1(r,c,i,j) &\rightarrow_{\text{dead}} \text{input\_image}(r+i, c+j)\\
S_1(r,c,i,j) &\rightarrow_{\text{dead}} \text{kernel}(i,j)\\
S_2(r,c) &\rightarrow_{\text{dead}} \text{output\_image}(\text{dead\_rows}+r, \text{dead\_cols}+c)\\
S_2(r,c) &\rightarrow_{\text{write}} \text{output\_image}(\text{dead\_rows}+r, \text{dead\_cols}+c)\\
S_2(r,c) &\rightarrow_{\text{dead}} \text{normal\_factor}()
\end{aligned}$$

We compiled the kernel with gcc 4.8.3, clang+Polly pre-3.6 (r226126) and icc 15.0. Using the flags `-O3` alone and in combination with automatic parallelization (`-ftree-parallelize-loops=2` (gcc), `-polly-parallel -lgomp` (clang + Polly) or `-parallel` (icc)) we automatically optimized and parallelized the code. Executing the benchmark on an Intel i7-3520M dual core system using an array of size 4096×4096 with a convolution kernel of size 3×3, we obtain the results shown in Figure 3e.

We can see that neither icc nor gcc is able to automatically parallelize the loop. Clang itself does not have any automatic parallelization facilities. However, in combination with Polly and our improved analysis, the loop nest is successfully parallelized and the execution time is reduced by 50%, now outperforming both icc and gcc. When optimizing the code in Listing 9, Polly generates the following run-time check:

$$KK \geq NN+1 \vee (NN \geq KK \wedge KK \geq \text{dead\_cols}+1 \wedge \text{dead\_cols} \geq 0)$$

The first condition under which the delinearization is valid is $KK \geq NN+1$. This is trivially true. If this condition is false, then the loop nest is not entered at all and there is consequently no access that can be out-of-bounds. The second condition verifies the validity of the delinearization in case the loop is actually executed. Theoretically, the source code shown provides enough information to statically prove the validity of the delinearization for all input values. However, LLVM's scalar evolution analysis does not support signed divisions, which prevents Polly from understanding that



(a) PolyBench/C – Speedup of Polly over 'clang -O3' using linear and delinearized arrays

Name	correlation	covariance	2mm	3mm	atax	bicg	cholesky	doitgen	gemm	gemver	gesummv	mvt	symm	syr2k	syrk
Parameter in subscript	+														
Run-time checks	+	+	+					+					+		
# 3D arrays								2							
# 2D arrays	2	2	5	7	1	1	1	1	3	1	2	1	3	3	2
# 1D arrays	2	1			3	4	13			8	3	4	5		

Name	trisolv	trmm	durbin	dynprog	gramschmidt	lu	ludcmp	floyd-warshall	reg_detect	adi	fdtd-2d	fdtd-apml	jacobi-1d-imper	jacobi-2d-imper	seidel-2d
Parameter in subscript			+	+				+		+					
Run-time checks									+						
# 3D arrays				1					2						
# 2D arrays	1	2	2	2	3	1		1	3	3	3			2	1
# 1D arrays	8		4	4	5						1		2		

(b) PolyBench/C – Statistics about the delinearized array views

Compiler	linear	delin.	Speedup
icc	2.2	-	-
gcc	2.2	-	-
clang	2.2	-	-
Polly	2.2	**1.2**	**1.8x**

(c) Run time [s] of dgemm written in boost::ublas

Type	linear	delin.	Speedup
single float	13	**3**	**4.3x**
double float	14	**3**	**4.6x**
i16	7	**2**	**3.5x**
i32	13	**3**	**4.3x**
i64	15	**3**	**5.0x**
i128	22	**5**	**4.4x**

(d) Run time [s] of different Julia gemm kernels

Compiler	O3	parallel	Speedup
icc	1.4	1.4	1.0x
gcc	1.0	1.0	1.0x
clang	1.4	-	-
Polly	1.4	**0.7**	**2.0x**

(e) Run time [s] of UTDSP edge detection kernel

Compiler	O3	parallel	Speedup
		(2/4/8)	
icc	2.5	2.5/2.5/2.5	1.0x
gcc	2.5	2.5/2.5/2.5	1.0x
clang	2.5	-	-
Polly	2.5	1.6/1.4/1.4	**1.6/1.8/1.8x**

(f) Run time [s] of MATLAB convolution kernel

Figure 3: Experimental Results

dead_cols = KK/2 holds; consequently, there is insufficient static information available to Polly to derive the validity of the delinearization. However, as our approach is optimistic, Polly emits run-time checks that still allow it to optimize the program. The basic validity conditions Polly derives require dead_cols to be positive and strictly smaller than KK, a condition that always holds for dead_cols = KK/2. Overall, we see that for the UTDSP edge detection benchmark our delinearization approach has enabled a complex transformation that showed significant speedups compared to use of competing production quality compilers. Even though shortcomings in LLVM's scalar evolution would have hindered a statically proven delinearization, our optimistic approach enabled us to verify the needed optimization at run-time.

5.5 MATLAB

MATLAB [9] is a high-level language primarily focused on numerical computations, and is widely used for signal and image processing, communications, control systems and computational finance. Besides directly executing the developed programs within MATLAB, MATLAB provides a facility, called MATLAB Coder, that allows the generation of independent C and C++ programs from an algorithm implemented in MATLAB.

To evaluate our work we use a simple convolution kernel implemented in MATLAB. Even though the original MATLAB code consists of a single call to a built-in function, the derived C code consists of more than 100 lines of comment-free auto-generated program code and includes four loop levels that use pointer addressing to model the multi-dimensional arrays. For reasons of brevity we cannot present the full loop nest, but present in Listing 10 the struct definition used to model the multi-dimensional arrays. As we can see, arrays in MATLAB use a manually implemented multi-dimensional array of parametric size, where accesses use a base pointer in combination with a possibly polynomial offset, requiring delinearization to perform precise dependence analysis.

For our evaluation we use the MATLAB generated C code, which is in its structure more complicated than the UTDSP code. Hence, we first perform some simplifications to ensure compilers are able to address the delinearization problem. This includes loop invariant code motion to avoid `Array->size[0]` style loads in the core loop, the removal of one unnecessary maximum computation in the access function, the consistent use of 64 bit integer types and the simplification of an if-condition.

We then compile the canonicalized C code with gcc 4.8.3, icc 15.0 and clang 3.6 pre (+ Polly) using '-O3' as well as automatic parallelization capabilities. The resulting binary, performs a 3x3 convolution on a 4096×4096 floating point matrix repeating it 5 times. The results (median over 5 runs) for running on an 8-core Intel Xeon E5430 CPU are shown in Figure 3f. We can see that if compiled sequentially, there is no performance difference with the different compilers. However, when generating parallel code, our delinearization algorithm has been able to recover the multi-dimensionality of the array and generated relevant run-time checks that allow the verification of the delinearization. Using this information Polly was able to compute precise data-dependences that prove that executing the loop nest in parallel is correct. Parallel execution reduces the execution time of the correlation kernel from 2.5 to 1.4 seconds and yields a 1.6 to 1.8x speedup. Even though we see improving performance due to parallel execution only for

```
    dead_rows = KK / 2; dead_cols = KK / 2;
    ...
    row = 0;
    output_image_ptr = output_image;
    output_image_ptr += (NN * dead_rows);
    for (r = 0; r < NN - KK + 1; r++) {
      output_image_offset = output_image_ptr;
      output_image_offset += dead_cols;
      col = 0;
      for (c = 0; c < NN - KK + 1; c++) {
        input_image_ptr = input_image;
        input_image_ptr += (NN * row);
        kernel_ptr = kernel;
S0:     *output_image_offset = 0;
        for (i = 0; i < KK; i++) {
          input_image_offset = input_image_ptr;
          input_image_offset += col;
          kernel_offset = kernel_ptr;
          for (j = 0; j < KK; j++) {
S1:         temp1 = *input_image_offset++;
S1:         temp2 = *kernel_offset++;
S1:         *output_image_offset += temp1 * temp2;
          }
          kernel_ptr += KK; input_image_ptr += NN;
        }
S2:     *output_image_offset =
           ((*output_image_offset)/ normal_factor);
        output_image_offset++; col++;
      }
      output_image_ptr += NN; row++;
    }
```

Listing 9: Core loop nest in UTDSP edge detection benchmark

```
struct emxArray_real_T {
    double *data; long *size; long allocatedSize;
    long numDimensions; boolean_T canFreeData;
};
```

Listing 10: MATLAB multi-dimensional array declaration

up to 4-cores, the analysis and transformation capabilities enabled by our delinearization will prove useful for future optimizations.

6. RELATED WORK

There have been previous efforts at delinearization, starting with the work of Maslov [7], who introduced delinearization to speed up dependence analysis for arrays of fixed size and also briefly discussed non-linear references arising from arrays of symbolic size. He required iteration spaces of rectangular shape, along with sufficient information to statically validate the delinearization. Loop induction variables can only appear in at most one dimension of a recovered array access. Maslov contributed a second approach in his work on polynomial constraint simplification [8], where delinearization in the context of triangular iteration spaces and possibly non-rectangular (triangular) arrays is discussed, but most of the other previously mentioned restrictions still apply. Simbürger and Größlinger [12] recently addressed delinearization in the context of Polly. Their approach is specific to dependence analysis and does not recover array sizes or even array subscripts. The use of advanced mathematical tools such as quantifier elimination techniques makes it hard to incorporate their techniques into a production compiler. Cierniak [2] presented a solution independent of dependence analysis, discussed delinearization for non-rectangular arrays and also provided on ideas how to unify the delinearization of multiple subscripts. He did not discuss a symbolic solution and he required each loop index to appear in index expressions of most one array dimension.

7. CONCLUSION

We have developed an approach to recovering a multi-dimensional array view from single-dimensional polynomial array access expressions. Multi-dimensional array shapes with sizes given as individual parameters, parameters times a constant, or parameters plus a constant are handled, with the first two cases also supporting the use of identical parameters for extents along multiple dimensions. Our approach can recover the multi-dimensional view for array accesses even in those cases where we cannot prove the validity statically. Instead, we provide a set of conditions that can be used to verify the validity of the multi-dimensional view at run-time. The approach has been partially implemented in the context of LLVM and Polly and this implementation has been used to evaluate the approach on kernels from Julia, blast::ublas, PolyBench/C, UTDSPs and MATLAB. With our optimistic approach to delinearization, we have been able to recover the multi-dimensional structure of data accesses in all these contexts and we have shown that the resulting increase in dependence analysis precision can *enable* large speedups. We have observed several kernels with more than 5.0x speedup compared to Clang, large improvements compared to the Julia compiler, and 1.8x improvement on the sequential execution of a standard dgemm kernel over icc. The new support for analysis of parametrically sized arrays significantly widens the set of compute kernels for which precise data dependences can be computed, thereby enabling much greater opportunities for application of optimizing transformations.

Acknowledgments

The LLVM community and especially Hal Finkel, Armin Größlinger and Andrew Trick provided important inputs. We received funding from Google Europe (Fellowship of Efficient Computing), Swissuniversities (Platform for Advanced Computing Initiative), the U.S. National Science Foundation (0811457, 0926127, 0926687, 1059417, 1321147 and 1440749), the U.S. Department of Energy (DE-SC0012489) and the Louisiana State University.

8. REFERENCES

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[2] M. Cierniak and W. Li. Recovering logical data and code structures. Technical report, 1995.

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

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

[5] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization (CGO)*, 2004.

[6] C. Lee and M. Stoodley. Utdsp benchmark suite, 1998.

[7] V. Maslov. Delinearization: An efficient way to break multiloop dependence equations. *SIGPLAN Not.*, 27(7):152–161, July 1992.

[8] V. Maslov and W. Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. In *Int. Conf. on Parallel and Vector Processing*, 1994.

[9] MATLAB. *version 8.4.0150421 (R2014b)*. The MathWorks Inc., Natick, Massachusetts, 2014.

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

[11] L.-N. Pouchet. PolyBench/C 3.2. `http://sourceforge.net/projects/polybench/`.

[12] A. Simbürger and A. Größlinger. On the variety of static control parts in real-world programs: from affine via multi-dimensional to polynomial and just-in-time. In *Proc. of the 4th Inter. Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, Jan. 2014.

[13] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software (ICMS'10)*, LNCS 6327, 2010.