On Recovering Multi-Dimensional Arrays in Polly

Tobias Grosser  
ETH Zürich  
tobias.grosser@inf.ethz.ch

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

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

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

ABSTRACT

Although many programs use multi-dimensional arrays, the multi-dimensional view of data is often not directly visible in the internal representation used by LLVM. In many situations, the only information available is an array base pointer and a single dimensional offset. For problems with parametric size, this offset is usually a multivariate polynomial that cannot be analyzed with integer linear programming (ILP) solvers and consequently impedes the computation of precise data dependences.

In this paper, we present an approach to recover the multi-dimensional nature of accesses to arrays of parametric size. In case of insufficient static information, the developed algorithm produces the necessary run-time conditions to validate the recovered multi-dimensional form. The access description obtained significantly simplifies the dependence checks, making previously polynomial dependence problems precisely solvable by a linear solver. Our approach has been evaluated using a number of benchmarks from polybench (C99), boost::ublas (C++) and Julia.

1. INTRODUCTION

Dense multi-dimensional arrays are data structures common to many compute problems. To allow compilers to perform interesting optimizations, it is often necessary to understand the multi-dimensional nature of these arrays. Unfortunately, while multi-dimensional arrays are native constructs in C99, Fortran or Julia [1], this information is often lost when translating such languages to a low-level compiler IR. In addition, many programming languages (e.g., C90 and C++) do not natively support variable size multi-dimensional arrays. In C90 or C++, users implement such arrays by creating their own classes, templates or macros, leaving the compiler without information about the multi-dimensionality of these accesses. As a result, accesses to such arrays are seen by the compiler as single-dimensional accesses that directly correspond to how the array is laid out in memory. We call this single-dimensional view the linearized view of an array. The process of recoverign the multi-dimensional view from the linearized view has been referred to as delinearization in the literature.

Assuming the original index expressions are affine, the linearized accesses can have different properties. For arrays of constant size, linearized expressions will contain large integer coefficients (the array sizes). Despite the presence of these coefficients the linearized access expression remains affine. We illustrate this with a simple example in [Listing 1].

As affine expressions can be precisely modeled with integer maps, data flow analysis based on integer maps (e.g., the one provided by isl [10]) will yield optimal results. However, in cases where the size of the array is parametric ([Listing 2]), this is no longer true. The expressions we obtain by linearizing accesses to arrays of parametric size may now contain multiplications between loop indexes and variables such as \( m \times i \). Performing dependence analysis on the “linearized” view of an array is a complex problem not supported by most existing dependence analysis tools.

void constantSize(float A[1024][4096]) {
    for (int i = 0; i < 1024; i++)
        for (int j = 0; j < 4096; j++)
            A[i][j] = i + j;
    // A[4096 i + j] = ... is affine
}

Listing 1: Multi-dimensional array of constant size

void parametricSize(float A[n][m]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            A[i][j] = i + j;
    // A[m i + j] = ... is polynomial
}

Listing 2: Multi-dimensional array of parametric size

We address this issue by presenting a new approach that for several important cases can derive from a given set of one dimensional multivariate polynomial array accesses an
equivalent multi-dimensional view in which all array index
equations are affine. As existing data dependence anal-
yses can precisely analyze the resulting arrays, we obtain
precise data dependency information for kernels with such
polylinear index expressions. In this paper we present the
following contributions:

• An approach to recovering the multi-dimensional view
of arrays of parametric size; this approach also han-
dles cases where statically proving that recovering the
multi-dimensional view is impossible.

• Increased data dependence analysis precision provided
by our general approach which is useful beyond depend-
ence analysis.

2. MOTIVATING EXAMPLE

Listing 3 shows a simple gemm kernel implemented with
C99 variable length arrays. When compiling the code with
Clang and analyzing the access to A as it is represented by
the compiler’s intermediate representation, the scalar evolu-
tions of LLVM [7] derives an expression equivalent to an access
A[i * p + k]. Neither the original dimension-
ality nor the size of the individual dimensions is preserved.

```c
void gemm(int n, int m, int p,
    float A[n][p], float B[p][m],
    float C[n][m]) {
    L1: for (int i = 0; i < n; i++)
        L2: for (int j = 0; j < m; j++)
            L3: for (int k = 0; k < p; ++k)
                C[i][j] += A[i][k] * B[k][j];
}
```

Listing 3: gemm in C99 using variable length arrays

To guide our approach to recovering the multi-dimensional
view, we take advantage of structural information provided
by the parametric array sizes. For multi-dimensional arrays
with affine access functions we observe that, after lineariz-
ing the accesses to such arrays, all parameters that appear
within products of loop induction variables and parameters
are derived from the sizes of the original array dimensions.
In the previous example the only such product is i * p and
the contained parameter p directly corresponds to the inner
dimension of the array A. So we can guess that the original
array has been declared as A[i][p] with access functions
A[i][k]. To verify our guess, we need to check that all pos-
sible uses of this access will remain within the bounds of the
assumed array shape. For the inner dimension this means
that for all possible values of j the access to the inner
dimension of B remains within bounds is not possible. Instead we require a run-time condition
B[size1] >= C[size1] to avoid out of bounds accesses. Only
in case B is large enough, our approach models the ac-
tual run-time behavior correctly. Even though this example
looks rather contrived it is in fact a very realistic exam-
ple. Increasing modularity inspired for example by the new
C++11/C++14 standards very often yields situations where
relations between parameters are not obvious. Having the
facilities to still be able to perform useful optimization is be-
coming more and more important for optimizing compilers.

```c
void gemm(struct 2DArray *A, struct 2DArray *B,
    struct 2DArray *C) {
    L1: for (int i = 0; i < SIZE0_2D(C); i++)
        L2: for (int j = 0; j < SIZE1_2D(C); j++)
            L3: for (int k = 0; k < SIZE1_2D(A); ++k)
                ACCESS_2D(C, i, j) +=
                    ACCESS_2D(A, i, k) * ACCESS_2D(B, k, j);
}
```

Listing 4: gemm with manual multi-dimensional arrays

3. PROBLEM STATEMENT

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

The view consists of 1) a multi-dimensional array defini-
tion (including the number of array dimensions and sizes for
all but the outermost dimension), 2) for each original array
access, a corresponding multi-dimensional access.

We also pose a set of additional requirements on the view
we derive:

• (R1) The number of array dimensions is minimal.

• (R2) The array sizes are minimal.

• (R3) The new access functions are affine in loop pa-
rameters and program parameters.

• (R4) For each array access, the memory location di-
rectly obtained from the linearized subscript expres-
sion and the memory location obtained from the multi-
dimensional array after lowering it using the derived
array sizes and assuming a row-major array layout
are identical for all loop iterators within the iteration
space.

• (R5) The array subscript expressions for all but the
outermost dimension are, for all iterations within the
iteration space, within the bounds of the multi-dimensional
array.

For cases where the multi-dimensional view cannot be proven
correct statically, we derive a multi-dimensional view as dis-
cussed above and provide a set of conditions under which
this view is valid.
Conditions $R1$ and $R2$ are there to ensure that no unnecessarily complicated array views are computed. $R3$ is necessary to ensure that we can represent the resulting access expressions as integer maps. $R4$ ensures that the multi-dimensional form of the array has the same access characteristics as the single dimensional array. $R5$ ensures together with $R4$ 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 iff 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, which again is necessary for the correct computation of data dependences.

### 4. ARRAYS OF PARAMETRIC SIZE

In this section we present an algorithmic approach to linearize a multivariate polynomial to a multi-dimensional array of the shape $A[P_0][P_1]...[P_{n-1}]$, $P_i \in \mathbb{P}$ with $\mathbb{P}$ referring to the set of program parameters. This means we obtain array shapes with the size of each dimension being defined by a single parameter and with multiple dimensions possibly sharing the same parameter.

#### 4.1 Basic algorithm

We first propose a basic algorithm consisting of the following four steps:

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

As a first step, we collect information about possible array size parameters. 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 a loop induction variable makes the expression non-affine. However, in case a parameter $P_i$ is an array size parameter, $P_i$ may be removed from the index expressions during linearization such that the original expression is turned into an access with affine subscript expressions. Consequently, we guess that $P_i$ 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 in the previous step and assume all of them form products. 

The code shown in Listing 5 initializes a subarray 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 $n_0 \times n_1 \times n_2$.

```c
void set_subarray(float A[],
    unsigned o0, unsigned o1, unsigned o2,
    unsigned s0, unsigned s1, unsigned s2,
    unsigned n0, unsigned n1, unsigned n2) {
  for (unsigned i = 0; i < s0; i++)
    for (unsigned j = 0; j < s1; j++)
      for (unsigned k = 0; k < s2; k++)
        A[(n2 (n1 o0 + o1) + o2) + n1 n2 i + n2 j + k] = 1;
}
```

Listing 5: Initialization of subarray

To recover the multi-dimensional nature of the access in statement $S$, we first expand the offset expression $(n_2(n_1 o_0 + o_1) + o_2) + n_1 n_2 i + n_2 j + k$ to a sum of products $n_2 n_1 o_0 + n_2 o_1 + o_2 + n_1 n_2 i + n_2 j + k$. 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 $(n_1 n_2, n_2)$. 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 $n_1 o_0 + o_1 + n_1 i + j$. 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 $A[i + o_0][j + o_1][k + o_2]$.

As a last step, we check the correctness of our linearization by forming the validity condition:

$$\forall i, j, k: \ 0 \leq i < s_0 \land 0 \leq j < s_1 \land 0 \leq k < s_2; \ 0 \leq k + o_2 < s_2 \land 0 \leq j + o_1 < n_1 \land 0 \leq i + o_0 \leq n_1 - 1 \land o_2 \leq$$

Using isl [10] we simplify this condition to $o_1 \leq n_1 - 1 \land o_2 \leq$...
Figure 1: Subarray accesses for different parameter values $n_2 - s_2$ 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 $n_0 = 8 \land n_1 = 9$. The first set of parameter values is $o_0 = 1 \land o_1 = 3 \land s_0 = 3 \land s_1 = 6$, which yields $3 \leq 9 - 6$ and evaluates to $T$. 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 $o_0 = 5 \land o_1 = 6 \land s_0 = 3 \land s_1 = 6$ (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 access data-locations that correspond to the array elements $A[i, j] : 4 \leq i \leq 6 \land 0 \leq j \leq 2$ (red squares). This is problematic, as e.g., the data stored to $A[7][9]$ affects the values read from $A[5][0]$. This relation is not visible in the delineralized program, which means the corresponding data dependences are not modeled and certain program transformations may be performed incorrectly. When checking our validity conditions we see that $o_1 \leq n_1 - s_1 \Rightarrow 6 \leq 9 - 6 \Rightarrow 1$, which correctly shows that for this set of parameters we cannot rely on our delinerealization.

### 4.2 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 delineralized using the same assumed array shape. Ensuring this requires only a slight adjustment of our 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 our analysis.

### 4.3 Subscripts containing size parameters

Listing 6 shows an example of multi-dimensional array accesses where the subscript expressions contain array size parameters. In case these accesses are linearized, as illustrated in Listing 7, the algorithm presented in Section 4 derives for an access $A[0][0][M - 1]$ with array size $A[i][j][k]$ the access $A[0][1][1] - 1$ as it associates multiples of $M$ with the second dimension. This delinerealization is invalid, as the subscript in the inner dimension becomes negative.

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

$$A[f_0][f_1] = A[f_0 + s_1 + f_1] = A[(f_0 - k)s_1 + (k+s_1 + f_1)] = A[f_0 - k][ks_1 + f_1]$$

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 address expressions. However, for a specific set of loop iterators and parameters not all $k_t$ values ensure in-bounds memory accesses. The question is how 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 access $A[i + 1][p]$ to an array of shape $A[1][N]$ in case $0 \leq p < N$ holds, but which would better be mod-
eled as $A[i][N + p]$ in case $N \leq p < 0$ 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., $-10 \leq p < N - 10$) there is not even a single optimal value for $k$. To still be able to model such cases we can create a piecewise 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 $(f_0, f_1) \rightarrow (f_0 + k, -k s_1 + f_1) | \exists k: k s_1 \leq f_1 < (k + 1) s_1$, 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 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.

\[
(f_0, f_1) \rightarrow \begin{cases} 
(f_0 - k_l, k_l s_1 + f_2) & f_1 < -k_l s_1 \\
(f_0 - 1, s_1 + f_2) & -s_1 \leq f_1 < 0 \\
(f_0, f_1) & 0 \leq f_1 < s_1 \\
(f_0 + 1, s_1 + f_2) & s_1 \leq f_1 < 2s_1 \\
(f_0 + k_u, -k_u s_1 + f_2) & k_u s_1 \leq f_1 
\end{cases}
\]

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 $(\ldots, f_t, f_{t+1}, \ldots) \rightarrow (\ldots, f_t + k, -k s_{t+1} + f_{t+1}, \ldots) | \exists k: k s_{t+1} \leq f_t < (k + 1) s_{t+1}$ using a finite set of affine pieces. Starting from the highest dimension $d$, we apply all maps $M_1$ one by one to the delinearized (i.e., the multi-dimensional view) accesses. Using the original algorithm on the example given in Listing 6 we obtain the following set of delinearized accesses: $\{S1(i, j, k) \rightarrow A(i, j, k), S2() \rightarrow A(1, 1, 1), S3() \rightarrow A(0, 1, -1), S4() \rightarrow A(1, -1, 0), S5() \rightarrow A(1, 0, -1)\}$. After applying a set of maps $M_t$ generated with variables $k_{t,l} = 0, k_{t,u} = 0$ 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:

\[
S1(i, j, k) \rightarrow \begin{cases} 
A(i - 1, N + j - 1, M + k) & k \leq -1 \land j \leq 0 \\
A(i, j - 1, M + k) & k \leq -1 \land j \geq 1 \\
A(i - 1, N + j, k) & k \geq 0 \land j \leq -1 \\
A(i, j, k) & k \geq 0 \land j \geq 0 
\end{cases}
\]

$S2() \rightarrow A(1, 1, 1), S3() \rightarrow A(0, 0, M - 1), S4() \rightarrow A(0, N - 1, 0), S5() \rightarrow A(0, N - 1, M - 1)$

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 for S1 the mapping $S1(i, j, k) \rightarrow A(i, j, k)$. So the piecewise mappings have all been statically reduced to maps with just a single piece.

### 4.4 Arrays of size $A[\cdot][\beta_1 P_1][\beta_2 P_2]$  

In certain cases (e.g., resizing of images) we may have array sizes of the form $A[\cdot][\beta_1 P_1][\beta_2 P_2], P_i \in P, \beta_i \in N$. 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 $A[\cdot][P_1][P_2]$. As $f_2$ can be in the range $0 \leq f_2 < \beta_2 P_2$, 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 on 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 positive in the sense that it only increases the chance that our 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.

## 5. Parameter + Constant  

We look now at a specific case, where the shape of the array is of the form $A[P_1 + \alpha_0] \ldots [P_{n-1} + \alpha_{n-1}]$ with $\forall i \in \{0, n - 1\}: P_i \in P, \alpha_i \in N$, with $P_i$ being different for different values of $i$. As an example we show in Listing 8 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.

```c
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][k+1] + In[i-1][j-1][k-1]);
```

Listing 8: Dimensions of size $P_i + \alpha_i$ (multi-dimensional)
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 we now present a general approach that allows us to delinearize polynomial expressions to d-dimensional array shapes of the form $A[P_1 + \alpha_1] \ldots [P_{d-1} + \alpha_{d-1}]$, $P_i \in \mathbb{P}$, $\alpha \in \mathbb{N}$, we look at the two and three dimensional special cases.

An access to a two dimensional array $A[f_0(\vec{i})][f_1(\vec{i})]$ with shape $A[|P_1 + \alpha_1|]$ corresponds to the single dimensional access $A[f_0(\vec{i}) + f_1(\vec{i})]$, which after expansion becomes $A[f_0(\vec{i})]P_1 + f_0(\vec{i})\alpha_1 + f_1(\vec{i})]$. However, it is unlikely that this structure is preserved. The only structure that can be assumed is a sum of terms $g_{[1]}(\vec{i})P_1 + g_0(\vec{i})$ where each term contains a different subset of the program parameters. In the general case we write this expression as

$$\sum_{\vec{r} \in \mathbb{P}([0,f-1])}(g_{\vec{r}}(\vec{i}) \prod_{s \in S} P_i),$$

where $f$ is the number of parameters, $\mathbb{P}([0, f - 1])$ is the power set of $[0, f - 1]$. The different $g_{\vec{r}}(\vec{i})$ are expressions in loop induction variables, and the different $P_i$ are program parameters. To delinearize this polynomial expression we need to recover from this sum expressions $f_0(\vec{i}), f_1(\vec{i}), \alpha_1$ as a function of $g_{\vec{r}}$. As $f_0(\vec{i})$ is the only coefficient to $P_1$, recovering the relation $f_0(\vec{i}) = g_{[1]}(\vec{i})$ is easy. The second equality we can obtain is $g_0(\vec{i}) = f_0(\vec{i})\alpha_1 + f_1(\vec{i})$. With $f_0(\vec{i})$ plugged in we obtain $g_0(\vec{i}) = g_{[1]}(\vec{i})\alpha_1 + f_1(\vec{i})$, which allows us to express $f_1(\vec{i})$ as a function of $\alpha_1$: $f_1(\vec{i}) = g_0(\vec{i}) - g_{[1]}(\vec{i})\alpha_1$. For different values of $\alpha_1$ 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 $\alpha_1$ to verify, it is possible to scan a certain number of $\alpha_1$ by either statically checking for valid delinearizations or possibly even by generating run-time version code for different values of $\alpha_1$.

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

$$f_0(\vec{i})P_1P_2 + f_0(\vec{i})P_1\alpha_2P_2 + f_0(\vec{i})P_2\alpha_1 + f_0(\vec{i})\alpha_1\alpha_2 + f_1(\vec{i})P_2 + f_1(\vec{i})\alpha_2 + f_2(\vec{i})$$

It corresponds to the polynomial expansion:

$$g_{[1,2]}(\vec{i})P_1P_2 + g_{[1]}(\vec{i})P_1 + g_{[2]}(\vec{i})P_2 + g_0(\vec{i})$$

From the single term that contains $P_1P_2$, the product of all symbolic parameters defining the array sizes, we recover $f_0(\vec{i}) = g_{[1,2]}(\vec{i})$. Assuming $P_1$ is the outermost parameter, we obtain the value of $\alpha_2$ from the single term that contains $P_1$, but not $P_2$: $g_{[1]}(\vec{i}) = f_0(\vec{i})\alpha_2 \Rightarrow \alpha_2 = g_{[1]}(\vec{i})/f_0(\vec{i}) = g_{[1]}(\vec{i})/g_{[1,2]}(\vec{i})$. Looking at the $P_2$ terms, we obtain the relation $g_{[2]}(\vec{i}) = f_0(\vec{i})\alpha_1 + f_1(\vec{i})$. This allows us to derive $f_1(\vec{i}) = g_{[2]}(\vec{i}) - f_0(\vec{i})\alpha_1 = g_{[2]}(\vec{i}) - g_{[1,2]}(\vec{i})\alpha_1$. Again, an expression containing $\alpha_1$ as a free variable. To obtain $f_2(\vec{i})$ we look at the terms without any parameters. Here we have $g_0(\vec{i}) = f_0(\vec{i})\alpha_1\alpha_2 + f_1(\vec{i})\alpha_2 + f_2(\vec{i})$ from which we can derive $f_2(\vec{i}) = g_0(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 - f_1(\vec{i})\alpha_2 = g_0(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 - (g_{[2]}(\vec{i}) - f_0(\vec{i})\alpha_1)\alpha_2 = g_0(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 - g_{[2]}(\vec{i})\alpha_2 + f_0(\vec{i})\alpha_1\alpha_2 = g_0(\vec{i}) - g_{[2]}(\vec{i})\alpha_2$. As $\alpha_1$ cancels out, we can unambiguously derive $f_2(\vec{i})$. We can conclude that delinearizing to a three-dimensional array shape does not introduce more freedom. Only $\alpha_1$ remains unknown and different values may need to be explored.

**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.

**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!

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 $f_0(\vec{i})$ and use it to try to derive a set of consistent $\alpha$ 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 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 polynomial expression we are analyzing. Specifically, that we can group it such that each term is the product between a subset of the suspected array size parameters and an expression $g_0(\vec{i})$ in loop indexes, non array size parameters and integer constants:

$$g_0(\vec{i}) + g_{[1]}(\vec{i})P_1 + g_{[2]}(\vec{i})P_2 + \cdots + g_{[d-1]}(\vec{i})P_{d-1} + g_{[1,2]}(\vec{i})P_1P_2 + g_{[1,3]}(\vec{i})P_1P_3 + \cdots + g_{[2,3]}(\vec{i})P_2P_3 + \cdots + g_{[1,d-1]}(\vec{i})P_1 \cdots P_{d-1} = \sum_{K \in \mathbb{P}([1,d-1])} (g_K(\vec{i}) \prod_{k \in K} P_k)$$

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

\[
\begin{align*}
& f_0(\vec{t})(P_1 + \alpha_1)(P_2 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\
& + f_1(\vec{t})(P_3 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\
& + \ldots + f_{d-2}(\vec{t})(P_{d-1} + \alpha_{d-1}) \\
& + f_{d-1}(\vec{t}) \\
& = \sum_{j \in [0,d-1]} \left( f_j(\vec{t}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \right)
\end{align*}
\]

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

\[
\begin{align*}
& \sum_{K \in \mathcal{P}(\{1,d-1\})} \left( g_k(\vec{t}) \prod_{k \in K} P_k \right) \\
& = \sum_{j \in [0,d-1]} \left( f_j(\vec{t}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \right) \\
& = \sum_{j \in [0,d-1]} \sum_{K \in \mathcal{P}(\{j+1,d-1\})} \left( f_j(\vec{t}) \prod_{k \in K} P_k \prod_{k \in [j+1,d-1]-K} \alpha_k \right)
\end{align*}
\]

We now equate terms that contain the same set of parameters, and, assuming the parameters to be positive, drop the common parameters on both sides. As a result, we obtain the following relations for \(g_k(\vec{t})\) and \(\alpha\) that we want to derive.

We first derive \(f_0(\vec{t}) = g_{[1,d-1]}(\vec{t})\), 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 \(g_k(\vec{t})\)

**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 \(g_{[1,d-1]}(\vec{t})\) not evenly divides \(g_{[1,d-1]\{k\}}(\vec{t})\) then
  \[\alpha_k = g_{[1,d-1]\{k\}}(\vec{t})/g_{[1,d-1]}(\vec{t})\]
  **return** 

  **foreach** \(S \in \mathcal{P}([2,k-1] \cup ([1,d-1] \{ k \}))\) **do**

    if \(g_{[1,d-1]}(\vec{t})\) not evenly divides \(S\) then
    \[\alpha'_k = g_{[1,d-1]}(\vec{t})/g_{[1,d-1]}(\vec{t})\]
    **return** 

  if \(\alpha'_k \neq \alpha_k\) then
    **return** 

**return** \(\{ k \to \alpha_k : k \in [2,d-1]\}\)

Then, we derive the values \(\alpha_k\) by looking at the terms \(g_{[1,d-1]\{k\}}(\vec{t})\). In the four-dimensional case such terms have the form:

\[
\begin{align*}
& g_{[2,3,4]}(\vec{t}) = \alpha_1 f_0(\vec{t}) + f_1(\vec{t}) \\
& g_{[1,3,4]}(\vec{t}) = \alpha_2 f_0(\vec{t}) \\
& g_{[1,2,4]}(\vec{t}) = \alpha_3 f_0(\vec{t}) \\
& g_{[1,2,3]}(\vec{t}) = \alpha_4 f_0(\vec{t})
\end{align*}
\]

In general \(\alpha_k, k \in [2,d-1]\) is \(\alpha_k = g_{[1,d-1]\{k\}}(\vec{t})/g_{[1,d-1]}(\vec{t})\).

Similar to the two and three-dimensional case, we cannot derive a value for \(\alpha_1\), as we do not know the value of \(f_1(\vec{t})\). However, for higher dimensional cases we can make an interesting observation. The values of \(\alpha_1\) 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 \(\alpha_k\) for an array view to be a valid delinearization. Specifically, to derive \(\alpha_k, k \in [1,d-1]\) we can choose any pairs of sets \((S,T), \emptyset \subset S \subseteq [1,k-1] \land T = S \cup \{k\}\) which can be used to compute \(\alpha_k\), as shown in Figure 2. The following lists the closed form expressions that compute \(\alpha_2\) and \(\alpha_3\) for the 4D case:

\[
\begin{align*}
& \alpha_2 = g_{[1,3,3]}(\vec{t})/g_{[1,2,3]}(\vec{t}) \\
& \alpha_3 = g_{[1,2,3]}(\vec{t})/g_{[1,3,3]}(\vec{t})
\end{align*}
\]

As the different values of \(\alpha\) are overdefined, we can use this information to cross check our delinearization. Specifically, we can use it to validate the order of the array size parameters. Algorithm 2 gives the full algorithm we use to obtain the different alpha values.

After having derived the different values of \(\alpha_k\), we can now derive the terms \(f_j, j \in [2,d-1]\) by looking at the terms \(g_{[j+1,d-1]}(\vec{t})\) (only interesting terms listed):

\[
\begin{align*}
& g_{[1,\ldots,d-1]}(\vec{t}) = f_0(\vec{t}) \\
& g_{[2,\ldots,d-1]}(\vec{t}) = \alpha_1 f_0(\vec{t}) + f_1(\vec{t}) \\
& g_{[3,\ldots,d-1]}(\vec{t}) = \alpha_1 \alpha_2 f_0(\vec{t}) + \alpha_2 f_1(\vec{t}) + f_2(\vec{t}) \\
& = \alpha_2 g_{[2,\ldots,d-1]}(\vec{t}) + f_2(\vec{t}) \\
& g_{[j,\ldots,d-1]}(\vec{t}) = \alpha_{j-1} g_{[j-1,\ldots,d-1]}(\vec{t}) + f_{j-1}(\vec{t}) \\
& g_{[0]}(\vec{t}) = \alpha_{d-1} g_{[d-1]}(\vec{t}) + f_{d-1}(\vec{t})
\end{align*}
\]

from which we derive \(f_j(\vec{t}) = g_{[j+1,d-1]}(\vec{t}) - \alpha_j g_{[j,d-1]}(\vec{t})\).

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 runtime conditions. This step is unchanged from Section 4.

**Algorithm 3: Derive subscript expressions**

**Data:** A dimensionality \(d\), a set of expressions \(g_k(\vec{t}), s \in \mathcal{P}([0,d-1])\), a set of values \(\alpha_k, k \in [2,d-1]\)

**Result:** A set of expressions \(f_k(\vec{t}), k \in [0,d-1]\)

\(f_0(\vec{t}) = g_{[1,d-1]}(\vec{t})\);

/* The next line assumes \(\alpha_1 = 0\). */

\(f_1(\vec{t}) = g_{[2,d-1]}(\vec{t})\);

**foreach** \(j \in [2,d-1]\) **do**

  \(f_j(\vec{t}) = g_{[j+1,d-1]}(\vec{t}) - \alpha_j g_{[j,d-1]}(\vec{t})\)

**return** \(\{ j \to f_j(\vec{t}) : j \in [0,d-1]\}\)

6. IMPLEMENTATION

We implemented Section 4 within LLVM and Polly [3]. In our implementation, LLVM’s scalar evolution analysis has been used to perform the transformation of index expressions necessary, for example, to extract the array size parameter candidates or to perform the division and remainder computations we use to obtain the subscript expressions. Besides the basic delinearization support, we also
implemented support for array size parameters in the index expressions (Section 4.3) as well as support for deriving a unique array shape for a set of accesses (Section 4.2). We also have full support for the generation of run-time conditions that validate our delinearization. For the generation of run-time conditions, we rely on a new AST generator developed as part of isl [10] and use its support to generate AST expressions from user-provided integer sets. This feature allowed 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 ask isl to remove any 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.

7. EXPERIMENTAL EVALUATION

7.1 C99 arrays in polybench

We tested the implementation of our delinearization on all 30 polybench 3.2 kernels [5] with the use of C99 variable length arrays enabled (-DPOLYBENCH_USE_C99_PROTO). From the 29 kernels currently detected by Polly (Polly skips floyd-warshall due to zero extends in the loop bounds introduced by the use of 32 bit induction variables), the multidimensional view of arrays can be correctly recovered in 27 of them; only two kernels (ludcmp, fkdtd-awmil) 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 5 is applicable and would allow us to extend Polly to handle these cases as well. It is also interesting to note that the Polybench code is written in a way that all delinearizations should be statically provable. Nevertheless our delinearization concluded that run-time checks are necessary for five benchmarks (correlation, covariance, 2mm, doigen, symm). Looking closer into why run-time checks are still generated we understand that in the original polybench source code certain parameters have been accidentally swapped in the array declarations and loop bounds, which 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 ‘squared’) 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 our optimized loop is used. Even though not foreseen, this example nicely shows the benefits of our approach to delinearization. Not only did it prevent a possible miscompilation of this code, 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.

7.2 Julia / boost::ublas

We also verified our delinearization in two other environments. We implemented a simple gemm kernel in boost::ublas, a blas library that uses C++ expression templates to generate efficient code. After extensive inlining, LLVM can remove the noise of the template instantiations and the matrix multiply kernel is exposed to Polly. After some trivial loop invariant code motion performed manually, our Polly combined with delinearization successfully detects this kernel and computes the necessary run-time checks. We implemented the same kernel in Julia [1], a dynamic high-level language for scientific computing. Similarly to the ublas example, after some simple loop invariant code motion performed manually, delinearizing the array accesses yields the expected results, run-time checks are emitted and the Julia kernel can be optimized with Polly. For matrices of size \(1024 \times 1024\) and type float, Polly’s default optimization (mostly loop tiling) already speeds up the computation from 15.3 to 2.6 seconds when run single-threaded on an Intel i7-3520M CPU with 128 KB L1, 512 KB L2 and 4096 KB L3 cache.

8. RELATED WORK

There have been previous approaches for delinearization starting with Maslov [5] who introduced delinearization to speed up dependence analysis by reducing the complexity of linear dependence analysis problems that result from multi-dimensional arrays of known size. In his work Maslov also briefly discussed how to handle non-linear array references as they arise from arrays with symbolic sizes. Maslov’s work requires the iteration space boundaries to be known, the iteration space itself being rectangular (and starting from zero) and the boundaries to be integer constants. He lifts the last restriction when extending his work to arrays with symbolic
sizes, but the iteration space is still required to be rectangular and of a size that allows the delinearization to be proven statically. Furthermore, Maslov requires that: "each resulting dimension must contain at least one variable and no variable can appear in more than one dimension". Maslov does not explore delinearization outside of the context of dependence analysis and does not address the problem of finding a consistent delinearization for a set of array references. Maslov contributed a second approach in his work on polynomial constraint simplification [6] where delinearization in the context of triangular iteration spaces and possibly non-rectangular (triangular) arrays is discussed. Even though the restrictions on the shape of the iteration space have been lifted, the iteration space is still required to statically prove the delinearization. Also, delinearization is again only applied in the context of dependence analysis. No approach for a consistent delinearization of a group of array references has been shown. His polynomial constraint simplification may suffer from an explosion of the number of constraints, in case of large fixed-size offsets between arrays. Also, the example shown in the paper is exploiting a special case where an induction variable is only used in a single array dimension. It is unclear how general his approach is. Simbürger and Größlinger [7] recently discussed delinearization within Polly using quantifier elimination, but again they focused on solving dependence analysis issues. Cierniak [1] presents a solution independent of dependence analysis, discusses delinearization for non-rectangular arrays and also provides ideas how to unify the delinearization of multiple subscripts. However, he does not discuss a symbolic solution and he requires that each loop index appears in at most one array dimension.

9. FUTURE WORK

Even though the approach presented in this paper is already very useful, there are still several interesting research questions. Analysing the approach in the context of a wider range of compilers, programming languages and programs will help us to better understand how widely this approach is applicable. Depending on empirical results, it may be interesting to generalize our approach to a larger set of array shapes (e.g. parameter + constant where multiple dimensions share the same parameter or even array sizes defined by arbitrary affine expressions).

We also believe that the construction of run-time conditions is worth further investigation. Interesting questions are how to generate efficiently run-time conditions for kernels with many memory references as well as how to minimize the number of run-time checks needed.

10. CONCLUSION

In this paper, we have developed an approach to recovering a multi-dimensional view from lower level linearized polynomial array access expressions for affine accesses of multi-dimensional arrays. 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 is able to correctly 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 recovery 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 and polybench, where we have seen promising results. The new support for analysis of parametrically sized arrays significantly widens the set of compute kernels for which precise data dependences can be computed and which as a result are more likely to be optimized successfully.

Acknowledgements

Tobias Grosser contributed parts of this work while affiliated with INRIA and funded through a Google Europe Fellowship of Efficient Computing. This work was also supported in parts by the US National Science Foundation through awards, 0811457, 0926127, 0926687, 1059417 and 1440749, by the U.S. Department of Energy (award DE-SC0012489), and by Louisiana State University. Furthermore, the LLVM community and especially Hal Finkel, Armin Größlinger and Andrew Trick provided important inputs.

11. REFERENCES