Lightweight Array Contraction by Trace-Based Polyhedral Analysis

Hugo Thievenaz, Keiji Kimura, Christophe Alias

Lightweight Array Contraction by Trace-Based Polyhedral Analysis*

Hugo Thievenaz[1], Keiji Kimura[2], and Christophe Alias[1]

[1] CNRS, ENS-Lyon, Inria, University of Lyon, France
{firstname}.{lastname}@ens-lyon.fr
[2] Waseda University, 3-4-1 Okubo, Shinjuku-ku Tokyo, 169-8555, Japan
{firstname}@waseda.jp

Abstract. Array contraction is a compilation optimization used to reduce memory consumption, by reducing the size of temporary arrays in a program while preserving its correctness. The usual approach to this problem is to perform a static analysis of the given program, creating overhead in the compilation cycle. In this work, we take a look at exploiting execution traces of programs of the polyhedral model, in order to infer reduced sizes for the temporary arrays used during calculations. We designed a four step process to reduce the storage requirements of a temporary array of a given scheduled program, in which we used an algorithm to deduce array access functions for which bounds are modulos of affine functions of parameters of the program. Our results show memory reductions of an order of magnitude on several benchmarks examples from PolyBench, a collection of programs from the polyhedral community. Execution time is compared to a baseline implementation of a static algorithm, and results show speed-up factors up to 20.

Keywords: Compilation · Array Contraction · Polyhedral Model · Dynamic Analysis · Memory Allocation

1 Introduction

The problem of temporary memory allocation is a challenge for programs meant to be running on platforms that have limited computing resources. Such temporary arrays manipulate results of intermediate computations that are disposed of at the end of the program. They are therefore sometimes oversized, when array cells are left unused and not overwritten by following computation despite their value no longer being used. Array contraction is a program transformation whose goal is to detect such unused array cells and replace a write to another cell to an unused one, in order to reduce the effective memory footprint of the array and shrink its maximum size, allocating less memory to the buffer while keeping program correctness intact.

Figure 1 introduces two direct applications of this method. Both the local memories used by the CPUs, and the buffer(s) used for communicating between

* Supported by Inria through the Polytrace exploratory action.

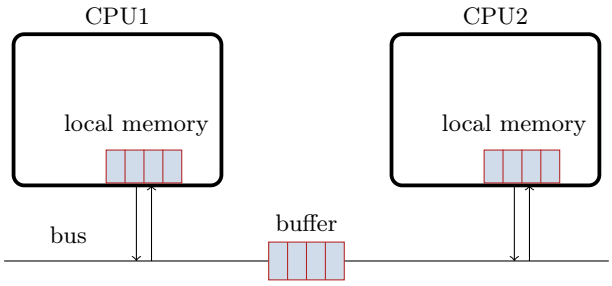

Fig. 1: Two CPUs perform computations on local memories and communicate data through a buffer

themselves, benefit from allocating smaller arrays. More precisely, many computing units require or may use on-the-fly memory sizing to exploit their structure (scratchpad memories, systems on chips). The class of programs studied for this optimization are affine loop kernels manipulating arrays (SCoPs, static control parts [2]), and this form of loop nest is the most common in many High-Performance Computing examples. The polyhedral model provides the necessary mathematical foundations to develop compiler optimizations such as array contraction, that focuses on compute-intensive scientific kernels containing such SCoPs. In this context, compilers rely on static analysis of the program to reduce the memory footprint of the program. Static analysis has been the basis of many works in the field [1,2,3,5,7,13]. However, dynamic analysis outclass static compiler ones when small execution traces can be efficiently produced and analysed. Static methods use polyhedral projections and Integer Linear Programming, which can be expensive depending on the shape of the code.

It would seem that no approach to this problem, to our knowledge, has explored the option of using dynamic analysis of the program in order to infer compilation optimizations. In this work, we contradict this habit and study the problem of determining a buffer allocation function from analysis of several execution traces. The problem can be formulated as follows: given a program manipulating a temporary array A, we want to infer allocations functions σ_A, of minimal image cardinal, such that any access $A[\boldsymbol{i}]$ can be safely replaced by $\hat{A}[\sigma_A(\boldsymbol{i})]$. Our general approach is then to apply a lightweight analysis on a few offline execution traces, with the assumption that the input parameter instances chosen for those traces are small enough that the execution time is significantly smaller.

In this paper, we make the following contributions:

– We present a new method for storage optimization, a dynamic approach that uses offline execution trace analysis. In particular, we describe a liveness algorithm from such execution trace, and another to compute the maximum number of variables alive alongside a dimension, from which we get our scalar modular mappings.

– We show, through the use of interpolation, that we can identify parameters from said modulo and deduce a generalized mapping.
– We implement this method on several benchmarks from the polyhedral community and show reductions both in implementation execution time and storage mappings deduced.

Our paper is structured as follows. Section 2 outlines the polyhedral model and the array contraction problem. Section 3 discusses related work. Section 4 describes our trace-based approach. Section 5 presents experimental results. Finally, Section 6 concludes this paper and draws research perspectives.

2 Background

We present the necessary background to the problem. We define the *polyhedral model*, and what is an usual polyhedral compilation flow. Then, we define the problem at hand, *array contraction*, and the related notions.

2.1 Polyhedral model

```
for (y=0; y<2; y++)
  for (x=0; x<N; x++)
S:  blurx[x,y] = in[x,y] +
      in[x+1,y] + in[x+2,y];
for (y=2; y<N; y++)
  for (x=0; x<N; x++)
T:  blurx[x,y] = in[x,y] +
      in[x+1,y] + in[x+2,y];
U:  out[x,y] = blurx[x,y-2] +
      blurx[x,y-1] + blurx[x,y];
```

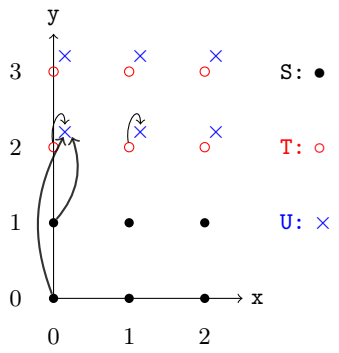

Fig. 2: Motivating example: 2D Blur filter

The *polyhedral model* defined by [8] is an intermediate representation of a loop nest as a graph over points of $\mathbb{Z}^n$. The class of programs that can be represented in this model, and therefore are subject to polyhedral optimizations, is polyhedral programs. These are (sequences of, possibly nested) *for* loops where all loop bounds and conditions are affine functions of the surrounding loop iterators and program parameters. Each execution of a statement `S`, nested in a n-depth loop, namely an *instance* or *operation*, can be represented by $\langle S, \boldsymbol{i} \rangle$ where $\boldsymbol{i}$ is a n-dimensional *iteration vector* of the surrounding loop indices. Its *iteration domain* D, the set of all possible iteration vectors for `S`, forms a graph over points of $\mathbb{Z}^n$.

Running example We illustrate our algorithm on the 2D Blur filter illustrated with its iteration domains on Figure 2. This is a well-known example of the polyhedral community, that applies two consecutive elementary convolutions on the input signal `in`. We have represented the iteration domains of `S`, `T` and `U` as colored symbols.

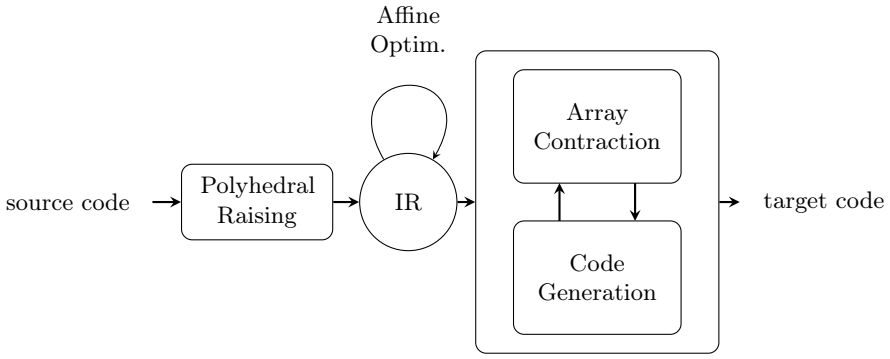

Fig. 3: Simplified polyhedral compilation flow of our method

Polyhedral compilation flow Compilation flows usually produce an in-between form of the program at hand, named Intermediate Representations (IR), on which transformations can be applied more easily to optimize its execution. Polyhedral compilation is no different, and in the case of our method, we produce those IRs as a pre-analysis step, separated from the algorithm. Figure 3 describes our simplified polyhedral compilation flow. Source code gets transformed once to an intermediate representation through *polyhedral raising*, who is then subject to possibly multiple affine transformations in order to optimize execution of the target code. The focus of our algorithm, and its performance, is therefore focused entirely on the application of such polyhedral optimization, for which ours is Array Contraction.

Affine transformations. At the heart of a polyhedral compiler, code transformations are expressed by affine mappings specifying a new execution order:

Definition 1 (Affine Scheduling). *A schedule maps each execution instance* $\langle S, \boldsymbol{i} \rangle$ *to an execution date* $\theta_S(\boldsymbol{i})$*. In the polyhedral model, schedules are affine per statement, i.e.* $\theta_S(i) = A_S i + b_S$*, and dates are vectors of* $\mathbb{Z}^p$ *ordered with the lexicographic ordering* $\ll$*. A schedule maps each iteration vector to its counterpart in the transformed, scheduled program.*

A possible schedule for our motivating example is the *canonical sequential schedule* θ, which is the order specified by the original `for` statements of the program: $\theta_S(y, x) = (0, y, x, 0), \theta_T(y, x) = (1, y, x, 0), \theta_U(y, x) = (1, y, x, 1)$.

Correctness. In the polyhedral model, data dependencies might be expressed between iterations. In the computation of the blur-interleaved example, for any instance where $y \geq 3$, we have that an instance of $\langle U, x, y\rangle$ of the second convolution depends (flow) on the preceding instances $\langle T, x, y\rangle$, $\langle T, x, y-1\rangle$, $\langle T, x, y-2\rangle$ of the first convolution. Anti- and output- dependencies are expressed in the same way. The dependence relation is denoted by $\rightarrow$. Of course, the schedule is constrained by data dependencies:

$$\langle S, \boldsymbol{i}\rangle \rightarrow \langle T, \boldsymbol{j}\rangle \Rightarrow \theta_S(\boldsymbol{i}) \ll \theta_T(\boldsymbol{j}) \tag{1}$$

This gives affine constraints which allow to compute affine schedules [8].

2.2 Array Contraction

The problem of array contraction, given a temporary array A, consists in finding a mapping $A[\boldsymbol{i}] \rightarrow \hat{A}[\sigma_A(\boldsymbol{i})]$ reducing or matching the unknown required size of A, minimal size for which the correctness of the program is intact. In our case, we seek memory mappings of the form $\sigma_A(\boldsymbol{i}) = \boldsymbol{i} \bmod \boldsymbol{b}(\boldsymbol{N})$, where b is an affine function of program's structure parameters $\boldsymbol{N}$ (e.g. array size).

Definition 2 (Conflict Relation). *A conflict relation $\bowtie_\theta$ is defined as the set of array cells whose lifetimes intersect during the execution of the program for the schedule θ.*

The conflict relation induces a correctness condition on array contraction, as conflicting array cells might be mapped to different places:

$$a[\boldsymbol{i}] \bowtie_\theta a[\boldsymbol{j}] \wedge \boldsymbol{i} \neq \boldsymbol{j} \Rightarrow \sigma_a(\boldsymbol{i}) \neq \sigma_a(\boldsymbol{j}) \tag{2}$$

Running example (cont'd) With the original loop schedule, the temporary array `blurx` might be contracted with the mapping $(y, x) \mapsto (y \bmod 3, x \bmod N)$, when $N \geq 3$. Indeed, `blurx` bufferizes the first convolution (`S`,`T`) before applying the next convolution (`U`) which only needs three rows y. This way, the footprint is reduced to $3 \times N$ array elements.

The *successive minima technique* [10] is the state-of-the-art approach to compute such mappings. The method of this work by Lefebvre and Feautrier can be boiled down to the following process. The conflict relation is represented as a difference set $\Delta_a = \{\boldsymbol{i}-\boldsymbol{j} \mid a[\boldsymbol{i}] \bowtie a[\boldsymbol{j}]\}$ for each array a; Then, for each array dimension k, the modulos are computed with $b_k(\boldsymbol{N}) = 1+\max\{\delta_k \mid (\delta_1, \ldots, \delta_n) \in \Delta_a\}$. Finally, resolved conflicts are removed before iterating on the next array dimension : $\Delta_a := \Delta_a \cap \{\boldsymbol{\delta} \mid \delta_k = 0\}$.

Our contribution, as we will show later, consists in a lightweight instantiation of this algorithm on several small execution traces, followed by an interpolation to obtain a general mapping. We show experimentally that *our results are obtained way faster than with the Lefebvre-Feautrier method, the latter dealing with costly parametric Integer Linear Programming (ILP).* More fundamentally, **this work is a proof-of-concept that costly polyhedral analysis might be rephrased as lightweight trace analysis. This opens new perspectives to scale polyhedral compilers**.

3 Related work

We quickly go over the multiple works related to our subject. We first present defining works on the subject. We then go over closely related work on the subject of array contraction. Finally, we present loosely related work on trace manipulation and analysis, but no work on dynamic array contraction by trace analysis has crossed our eyes.

Affine array contraction As described by [3,10], and again in this work, the successive modulo technique seeks to reduce the memory storage requirements of an already scheduled program, by performing static analysis in order to construct a conflict set. The array dimensions are reduced by finding contraction moduli along the array's axes. While recalling that the method of Lefebvre and Feautrier [10] obtains on the example *blur-interleaved* a storage mapping $(y, x) \mapsto (y \bmod 3, x \bmod N)$, the more advanced work of Bhaskaracharya et al. [3] infers a more refined mapping $(y, x) \mapsto 2x - y \bmod (2N + 1)$, because their approach consider the change to a better basis for the contraction vectors.

Inter-array and intra-array contraction The type of optimization we are looking for in this paper is an *intra-array optimization* as designed by [3], and references such as [1,8,11] focus on this intra-array analysis. This means that the analysis performed is done on a per-array basis. [3,4] build a technique for intra-array as well as inter-array optimization, a technique that consider the reduction of multiple temporary arrays, allowing them to find even more reduced mapping by changing (often reducing) the dimensionality of the array(s) considered for the analysis. [6] calculate the memory requirements of a program by approaching them as a polynomials in the parameters of the program, but their method has to relax the solutions as rational instead of integer, and only give an upper bound of the memory consumption.

Trace analysis In terms of trace analysis, some work has already covered similar topics such as loop recognition and trace prediction [9] and trace-based affine loop reconstruction [12]. The former compresses traces (as sequences of scalars) and constructs a loop nest producing such sequence of numbers. The latter focuses on reconstructing loops based on their predictable affine behavior, from the addresses of the memory accesses, and presents a terminating algorithm to reconstruct the loop function entirely. These works, therefore, focus only on rebuilding incomplete traces, and not on the usage of traces in a compilation process.

4 Our approach

This section presents the contributions made to the problem of array contraction, and detail our method of offline trace execution and analysis to infer a general mapping.

4.1 Overview of the approach

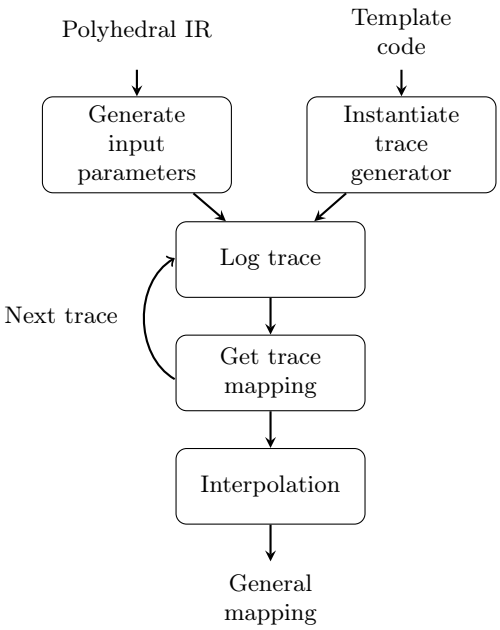

Fig. 4: Our approach

Figure 4 depicts our approach. We start with an input Intermediate Representation, comprised of the program, its schedule θ, and its Dependence Graph. We also input the code generated from θ (Template code), which will allow to produce traces.

First, we compute the input parameter instances on which running the program to obtain interesting traces. We also instrument the template code to prepare the trace generation. Then, for each input parameter instance $\boldsymbol{N}$, we generate the trace (Log trace) and we apply a *lightweight instance* of the successive minima method (Get trace mapping). We end up with a collection of trace mappings. Finally, we infer the general mapping (working for any input parameter) from an interpolation between each input parameter instance and its corresponding output modulo mapping (Interpolation). All these steps are detailed thereafter.

4.2 Generating input parameter instances

Our trace analysis operates on execution traces of programs, meaning we have to instantiate our kernel program with scalar values for its parameters $\boldsymbol{N}$. Since we

would like to interpolate modulos as an affine forms of parameters $\boldsymbol{N} \mapsto b_k(\boldsymbol{N})$, we need $|\boldsymbol{N}| + 1$ parameter instances. Also, the parameter instances must be independent to enforce a unique interpolation. We first explain how the first parameter instance is computed. Then, we explain how we get the remaining parameter instances.

First parameter instance, $\mathcal{O}$ We derive the set of parameters covering all the dependencies by projecting the dependency constraints of the program on each of the parameters. Then, we compute a minimum value for each of the parameters. Usually the constraints are of the form $N \geq \ell$ with ℓ some constant lower bound. Hence, we may infer a lower bound for each parameter with a simple syntactic heuristic without using expensive linear programming techniques. This gives the first parameter instance, denoted as $\mathcal{O}$ (for "origin").

This heuristic assumes that in the main program execution, all the dependencies are reached. However, this is not always the case, and if we deal with imperfect loop nests, then the result is an upper bound, giving potential additional overhead by operating on traces with bigger parameter instances than needed. How to extend this heuristic to the general case is left to future work. In our running example, the intersection of all the dependence constraints boils down to the polyhedron $\{N|N \geq 3\}$. Hence $\mathcal{O} = (3)$, denoting the parameter instance $N = 3$.

Remaining parameter instances One set of parameter values is not sufficient to establish an interpolation. For each parameter N_i, $1 \leq i \leq p$, we create a new parameter instance, linearly independent from the rest. A straightforward way to compute such new instances is to build the set of increments by each canonical vector $\boldsymbol{e}_i$: $\mathcal{I} = (\mathcal{O}, \mathcal{O} + e_1, \ldots, \mathcal{O} + e_p)$, where $\mathcal{O} + \boldsymbol{e}_i = (N_1, ..., N_i + 1, ..., N_p)$. These parameter instances will lead to a unique affine interpolation, as we will see later. On our example, we would obtain $(3, 4)$, denoting the parameter instances $N = 3$ and $N = 4$.

4.3 Inferring a mapping on a trace

The following algorithm 1 describes our lightweight instance of the successive minima method to compute a mapping from an execution trace.

We apply a direct liveness analysis on the trace to compute the difference set Δ_a we defined earlier, then each modulo is computed as the maximum difference measured alongside each array dimension i, following the lines of the successive minima algorithm. Since we deal with finite (and small) integer sets, no ILP is required.

This method is an instance of the Lefebvre-Feautrier algorithm [10] since we produce the same mapping while operating on a trace, i.e. an instantiated program. Since the mapping is affine, we can directly apply a linear interpolation to deduce a generalized parametrized mapping, as we will describe in the next section.

Algorithm 1: Find the mapping for the array a on the trace T

Result: mapping $\boldsymbol{i} \mapsto \boldsymbol{i} \bmod \boldsymbol{m}$
function GetMapping(T,a)
$(In, Out) \leftarrow$ liveness(T)
$CS \leftarrow \bigcup_p \{(a[\boldsymbol{i}], a[\boldsymbol{j}]) \mid a[\boldsymbol{i}], a[\boldsymbol{j}] \in In(p)\}$
$\Delta_a \leftarrow \{\boldsymbol{i} - \boldsymbol{j} \mid (a[\boldsymbol{i}], a[\boldsymbol{j}]) \in CS\}$
for *each array dimension i, starting from 0, in increasing order* **do**
  $m_i \leftarrow 1 + max\{\delta_i \mid (0, ..., 0, \delta_i, ...) \in \Delta_a\}$
end

Running example (cont'd) For our blur-interleaved example, such analysis would show, on the trace for $N = 3$, for the `blurx` array, that the biggest width alongside the `y` axis is 2, so there are a maximum of 2+1 array cells in conflict at any given control point p. The observation on `x` is similarly done, and again we measure a width of 2, and so a number of conflicts of 3. Hence, we obtain $(y, x) \mapsto (y \bmod 3, x \bmod 3)$. This is repeated for the trace with $N = 4$ where we obtain $(y, x) \mapsto (y \bmod 3, x \bmod 4)$.

4.4 Interpolation

From the mapping instances deduced, we show how to interpolate a generalized mapping that depends on program parameters. We retrieve mappings of the form $\boldsymbol{i} \mapsto \boldsymbol{i} \bmod \boldsymbol{b}(\boldsymbol{N})$ by a direct affine interpolation from the pairs of inputs (parameter instances) and outputs (modulo scalars found). We realise this by solving the following systems of equations. Let p be the number of program parameters ($\boldsymbol{N} = (N_1, \ldots, N_p)$), and k the number of array indices ($\boldsymbol{i} = (i_1, \ldots, i_k)$). Then:

$$\sigma_a(\boldsymbol{i}) = \begin{pmatrix} i_1 \bmod f_1(N_1, ..., N_p) \\ \vdots \\ i_k \bmod f_k(N_1, ..., N_p) \end{pmatrix} \tag{3}$$

This system of equation (3) defines the f_ℓ functions that we are determining. Expecting to deal with affine functions, each f_ℓ can be written in the homogeneous form:

$$f_\ell(\boldsymbol{N}) = \boldsymbol{\tau}_\ell \cdot \begin{pmatrix} \boldsymbol{N} \\ 1 \end{pmatrix} \tag{4}$$

where $\boldsymbol{\tau}_\ell$ is a vector of size $p + 1$, τ_i being the coefficient of N_i, for $1 \leq i \leq p$, and τ_{p+1} being the constant coefficient.

We have collected sample values from these affine functions, for each array index ℓ, represented by:

$$\begin{cases} f_\ell(\mathcal{O}) = m_0 \\ \vdots \\ f_\ell(\mathcal{O} + e_p) = m_p \end{cases} \tag{5}$$

Which can be written as $A\boldsymbol{\tau}_\ell = \boldsymbol{m}$:

$$\begin{pmatrix} \mathcal{O} & 1 \\ \mathcal{O}+e_1 & 1 \\ \vdots & \vdots \\ \mathcal{O}+e_p & 1 \end{pmatrix} \boldsymbol{\tau}_\ell = \begin{pmatrix} m_0 \\ \vdots \\ m_p \end{pmatrix} \tag{6}$$

We now show that this system has always a unique solution in $\mathbb{Z}^{p+1}$:

Theorem 1. *A is unimodular.*

Proof. We apply the Gaussian elimination method to express the determinant of A. We may substract from each of the first p columns that we label each a_i, the last column a_{p+1} multiplied by N_i without changing the determinant. The resulting matrix is as such:

$$\det A = \begin{vmatrix} 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & 1 \end{vmatrix}$$

It immediately follows that the determinant of this matrix is $(-1)^{p+1} \times \det I_p$, the permutation of the $p+1$-th and the first column leading to the $(-1)^{p+1}$ factor, and $\det I_p$ the determinant of the lower-left matrix which is the identity. Therefore, $\det A = \pm 1$ and so A is unimodular. □

Because A is unimodular, the linear equation system always has integer solutions. Therefore, for any given program with p parameters, $p+1$ traces are both necessary and sufficient to produce an interpolation.

Running example (cont'd) On traces, we obtained the *trace mappings* $(y, x) \mapsto (y \bmod 3, x \bmod 3)$ for $N = 3$, and $(y, x) \mapsto (y \bmod 3, x \bmod 4)$ for $N = 4$. Denoting $(y, x) \mapsto (y \bmod b_1(N), x \bmod b_2(N))$ the general mapping, we have $b_1(3) = 3$ and $b_1(4) = 3$. Hence we solve: $A\boldsymbol{\tau}_1 = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix} \boldsymbol{\tau}_1 = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$ from which we deduce $\boldsymbol{\tau}_1 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$. Hence $b_1(N) = \boldsymbol{\tau}_1 \cdot \begin{pmatrix} N \\ 1 \end{pmatrix} = 3$.

Also, $b_2(3) = 3$ and $b_2(4) = 4$. Hence we solve: $A\boldsymbol{\tau}_2 = \begin{pmatrix} 3 & 1 \\ 4 & 1 \end{pmatrix} \boldsymbol{\tau}_2 = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ from which we deduce $\boldsymbol{\tau}_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Hence $b_2(N) = \boldsymbol{\tau}_2 \cdot \begin{pmatrix} N \\ 1 \end{pmatrix} = N$.
Hence, we get the *parametrized program mapping* $(y, x) \mapsto (y \bmod 3, x \bmod N)$.

5 Experimental Results

This section presents our implementation and the results obtained with our approach, and make a comparison with the successive minima approach.

5.1 Experimental setup

We have implemented our method as an automatic code generator in C++ named PoLi. Our tool takes as input an intermediate representation of a kernel and first outputs a C program where its statements have been swapped out with calls to trace generation methods. Then, the lifetime analysis is performed on several execution traces, from a remote compilation and execution of the modified kernel. Finally, the deduced mapping is directly applied by modifying the access functions of the temporary arrays to the ones deduced.

The baseline implementation, used to compare our analysis time and storage requirement measurements with, is an implementation of the successive modulo technique [10]. The C kernels have been compiled using gcc 9.3.0 with flags "-fPIC -O3", while the implementation itself has been compiled using g++ 9.3.0 using flags "-O3 -ldl -lstdc++fs". Every compilation and execution of the kernels, and so their time measurements have been done on an Intel Core i5-1135G7 CPU running at 2.40GHz. No HPC computer is required, as we deal with *compilation*, not execution. The LF method is compiled with the same directives. We list the examples present in our benchmarks, which are part of the PolyBench test suite[3]:

- **fibonacci** computes the n-th term of the fibonacci sequence. It showcases very low runtime because of a very simple single loop nest.
- **pc-2d** and **pc-2d-line**, two examples of a producer-consumer mechanic in two dimensions, respectively without and with the last 2 rows of `A` explicitly being output dependencies. Those show the relevance of the method to only temporary memory.
- **blur-interleaved** and **blur-tiled**, two examples of the 2D blur filter, respectively with producer-consumer statements interleaved (motivating example), and tiled scheduling. Together, they highlight the versatility of the method, matching the Lefebvre-Feautrier approach for the interleaved case, but outpaces it when the loop nest gets more complex with more loop counters added for the tiling.
- **2mm**, example of two successive matrix multiplication and assignment. This example shows that the Lefebvre-Feautrier method also suffers from the multiplicity of arrays in the program, which skyrockets its runtime compared to our approach.

5.2 Results

Table 1 depicts the kernels and their targeted temporary array, alongside its original size, and the mapping found is the reduced size inferred from our algorithm. Parameter instance represents the starting parameter values chosen for the analysis. The execution times shown are not the ones of the modified kernels' executions, as the mappings found are the same for both methods. Rather, the first average runtime describes, for our method PoLi, the sum of the measured

[3] available at https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

Kernel	Mapping found	Parameters	PoLi time (ms)	LF time (ms)	**Speed-up**
fibonacci	$i \bmod 2$	$N = 2, 3$	0.00103	0.024221	**23.5**
pc-2d	$i \bmod N$ $j \bmod N$	$N = 2, 3$	0.00284	0.045513	**16.0**
pc-2d-line	$i \bmod 2$ $j \bmod N$	$N = 3, 4$	0.01022	0.064114	**6.3**
blur-2d	$y \bmod 3$ $x \bmod N$	$N = 5, 6$	0.15636	0.187037	**1.2**
blur-tiled	$y \bmod 3$ $x \bmod 4$	$N = 5, 6$	0.166067	4.041242	**24.3**
2mm	$i \bmod N$ $j \bmod N$	$N = 2, 3$	0.096936	2.228872	**23.0**

Table 1: Mappings and runtimes obtained using our approach (PoLi) compared to the baseline successive modulo method (LF) [10]

time spent on the generation of the parameter instance, the time spent instantiating the trace and the time spent interpolating the resulting mappings. This is compared to the baseline runtime which represents the time spent applying the instance of the Lefebvre-Feautrier approach we have implemented, and we show the speed-up factor between the two methods ran successively. We can observe that the `fibonacci` example has a speedup of more than 20, explained by the small trace parameters chosen, as the runtime of PoLi on this example is noticeably the lowest out of all. `blur-tiled` and `2mm` have respectively bigger iteration dimension and a greater overall number of arrays, meaning the Lefebvre-Feautrier approach irremediably takes more time projecting over those several dimensions, whereas our method takes advantage of the smallness of the parameter instances selected and suffers way less from more arrays and array dimensions. The complexity of the LF method is directly tied to the iteration dimension in an exponential fashion, while our approach is less sensitive to it. `blur-interleaved` and `pc-2d-line` both present smaller speed-up factors, as our parameter instance generation gives an upper bound too big, while the dependencies can still be respected with lower parameter values. Therefore, more carefulness is required in the selection of the starting parameter instance, meaning that a better method to infer parameter instances is also of the essence. On these two examples, our approach still manages to match or outperform the Lefebvre-Feautrier method while having unnecessarily large starting parameter instances.

6 Conclusion

In this paper, we have presented a novel lightweight method for array contraction in the polyhedral model. This work is the very first step towards a new paradigm of trace-based analysis to scale polyhedral compilation and demonstrate a promising proof of concept on the array contraction problem. We design

and implement an automatic array contraction tool, that takes as input the source code of the kernel and outputs optimized target code in regards to storage space consumption. We present the algorithms and methodology used in our tool. Execution times are compared to those of the Lefebvre and Feautrier method and shows promising speed-up factors. Results answers positively to the question of the possibility of generalization from a subset of execution traces.

In the future, we seek to apply another methodology to the starting parameter instance deduction, in order to choose minimal parameters regardless of the form of the loop. We also look forward to deduce more complex mappings, of the form $\boldsymbol{i} \mapsto A\boldsymbol{i} \bmod \boldsymbol{b}(\boldsymbol{N})$, similarly to [1,3]. More generally, we seek to apply the paradigm of trace analysis to other problems of the polyhedral compilation, to further study the potential yield of trace analysis in the compilation process. Finally, we plan to address the interaction of array contraction and other optimizations passes by integrating our implementation in an automatic parallelizer.

References

1. Alias, C., Baray, F., Darte, A.: Bee+ cl@ k: An implementation of lattice-based array contraction in the source-to-source translator rose. ACM SIGPLAN Notices **42**(7), 73–82 (2007)
2. Bastoul, C., Cohen, A., Girbal, S., Sharma, S., Temam, O.: Putting polyhedral loop transformations to work. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 209–225. Springer (2003)
3. Bhaskaracharya, S.G., Bondhugula, U., Cohen, A.: Automatic storage optimization for arrays. ACM Transactions on Programming Languages and Systems (TOPLAS) **38**(3), 1–23 (2016)
4. Bhaskaracharya, S.G., Bondhugula, U., Cohen, A.: Smo: An integrated approach to intra-array and inter-array storage optimization. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 526–538 (2016)
5. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113 (2008)
6. Clauss, P., Fernández, F.J., Garbervetsky, D., Verdoolaege, S.: Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. IEEE transactions on very large scale integration (VLSI) systems **17**(8), 983–996 (2009)
7. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. IEEE Transactions on Computers **54**(10), 1242–1257 (2005)
8. Feautrier, P., Lengauer, C.: Polyhedron model. Encyclopedia of parallel computing **1**, 1581–1592 (2011)
9. Ketterlin, A., Clauss, P.: Prediction and trace compression of data access addresses through nested loop recognition. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization. pp. 94–103 (2008)
10. Lefebvre, V., Feautrier, P.: Automatic storage management for parallel programs. Parallel computing **24**(3-4), 649–671 (1998)

11. Quilleré, F., Rajopadhye, S.: Optimizing memory usage in the polyhedral model. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(5), 773–815 (2000)
12. Rodríguez, G., Andión, J.M., Kandemir, M.T., Touriño, J.: Trace-based affine reconstruction of codes. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 139–149 (2016)
13. Simbürger, A., Apel, S., Größlinger, A., Lengauer, C.: Polyjit: Polyhedral optimization just in time. International Journal of Parallel Programming **47**(5), 874–906 (2019)