Towards Trace-Based Array Contraction

Hugo Thievenaz, Keiji Kimura, Christophe Alias

Towards Trace-Based Array Contraction

Hugo Thievenaz, Keiji Kimura, Christophe Alias

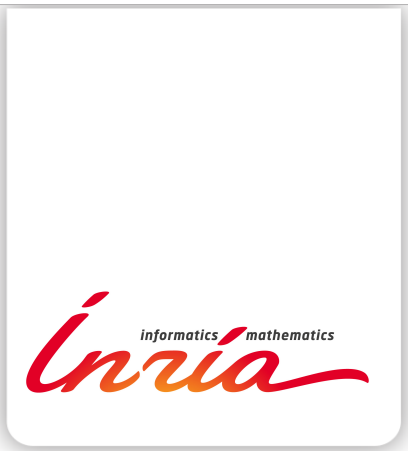
RESEARCH REPORT

N° 9442

Décembre 2021

Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9442--FR+ENG



Towards Trace-Based Array Contraction

Hugo Thievenaz*, Keiji Kimura†, Christophe Alias‡

Project-Team Cash

Research Report n° 9442 — Décembre 2021 — 17 pages

Abstract: Array contraction is a compilation optimization used to reduce the memory consumption, by shrinking the size of temporary arrays while preserving the 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 report, 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 five 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 and counters of the program. Our preliminary results show reductions of an order of magnitude on several benchmarks examples from the polyhedral community.

Key-words: Compilation, array contraction, polyhedral model, dynamic analysis

* Inria/ENS-Lyon/UCBL/CNRS
† Waseda University
‡ Inria/ENS-Lyon/UCBL/CNRS

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Vers une contraction de tableaux par analyse de traces

Résumé : La contraction de tableau est une optimisation de compilation servant à amoindrir les coûts en mémoire, en réduisant la taille des tableaux temporaires sans en altérer l'exactitude du résultat. L'approche habituelle pour ce problème est l'analyse statique du program, ce qui engendre plus de travail dans le cycle de compilation. Nous étudions les traces d'exécution de programmes du modèle polyédrique, afin d'en inférer des tailles réduites pour ces tableaux temporaires. Nous proposons une méthode en cinq étapes pour réaliser la contraction de tableaux sur un programme déjà ordonné, comprenant l'utilisation d'un algorithme pour déduire des fonctions d'accès aux tableaux affines. Nos résultats préliminaires comprennent des réductions d'un ordre de grandeur sur plusieurs exemples de la communauté polyédrique.

Mots-clés : Compilation, contraction de tableaux, modèle polyédrique, analyse dynamique

1 Introduction

Efficient temporary memory allocation is an important factor for programs meant to be running on platforms that have limited computing ressources, and for running time optimization in general if the memory footprint reduction allows to reduce cache misses. For some domains where heavy -but often predictable- data manipulation on limited resources is of the essence, such as stencil computation and image processing applications, compilers rely on static analysis of the program to reduce the memory footprint of the program. Dynamic analysis sometimes manages to outclass static compiler ones, as the availability of dynamic informations make some optimizations possible, and most, if not all static approaches use projections over $\mathbb{Z}^{\mathbb{N}}$ for their algorithms, which are expensive. Runtimes do however come with an overhead, making scheduling lean towards being *coarse-grain*, while compilers that map tasks to specific computation units seek to obtain *fine-grain* parallelism. Therefore, we detach from the scheduling problem and consider our input programs as scheduled already.

The class of statements considered for this work are those of affine loop nests, sequences of (non-)nested (im-)perfect loops where all accesses and loop bounds are affine functions of the surrounding loop iterators and program parameters. This restriction allows us to manipulate an intermediate representation of such programs known as *polyhedral programs*, and exploit both logical and geometrical optimizations. This has been the basis of many works in the field [1–3, 5, 6, 12], that all use static analysis in their method.

It would seem that no approach to this problem, to our knowledge, has explored the route of using dynamic analysis of the program on smaller inputs. In this paper, 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 $\mathtt{a}$, we want to infer allocations functions σ, of minimal image cardinal, such that any access to $\mathtt{a}[\overrightarrow{i}]$ can be safely replaced by $\mathtt{a}[\sigma(\overrightarrow{i})]$.

Our contributions to this problems are the following:

- We present a new method for storage optimization, a dynamic approach that uses offline execution trace analysis, and formulate the problem as finding contraction moduli along a static basis on an already scheduled program.
- We describe a liveness algorithm from an 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 moduli and deduce a generalized mapping.
- We implement this method on three benchmarks from the polyhedral community and show reductions in storage requirements.

1.1 Array contraction: an example

Say you want to compute, by hand, the n-th term of the Fibonacci sequence. Its recursive definition tells us that we need to compute the two previous terms first, and so on. For each new computation done, the only two values you look up are the two previous ones, and you no longer have any need for the older values; You could decide to scratch the oldest value at the end of each computation. Note that the goal is not for the computation to return the whole Fibonacci sequence up to n, but rather the n-th number of the sequence; Otherwise you would need to keep all the previous results around, because together they form your expected result, the entirety of the sequence up to n.

Algorithmically, the approach would be to express this problem and implement a solution in a framework that can describe linear structures of data, such as arrays and queues, in order to tie indices to your values. The temporary memory of a program described in such a framework would be the set of values (and so, their associated index) that have been computed and explicitely stored during computation, but are not relevant to the result. In this context, because the schedule is set in stone, the optimal size for the temporary array of your program is then the maximum number of variables that have not yet been read for the last time at any point of the execution; If we go lower than that, at least one statement will not be able to correctly execute, because one or more temporary values will have been overwritten before their last read. Let us consider the execution of the loop in this example.

```
fib[0] = 1;
fib[1] = 1;
for(i=2; i<=n; i++)
  fib[i] = fib[i-1] + fib[i-2];
result = fib[n];
```

Unrolling this loop for a given n gives us a sequence of $n+1$ statements, of which $n-2$ of them describe the computations done in the *for* loop. They are ordered, or more precisely *scheduled* in lexicographical order on i, and this is the only valid schedule for this computation, because iteration i depends on iteration $i-1$. Again, if we assume the purpose of this program is to return the n-th Fibonnaci number, then the `fib` array describes temporary memory, and the variable `result` corresponds to the *live-out* dependency of this program; it is a value required for what will follow of the computation. This is not true for the values of `fib`; they are only useful for the computations leading to the assignment of the result. For now, under those parameters of precise scheduling and no modification of the memory allocation of `fib`, there are at maximum $n+1$ variables live at the same time: the n values of `fib`, and the `result`; and this happens at the last statement. Now, since the only live-out value is `result`, we do not care about the final state of `fib`. Recall the observations made earlier when computing the sequence by hand: the only two values necessary to compute the i-th Fibonacci number are the two previous ones; Therefore, we should be able to reduce the size of the `fib` array down to 2, because for any statement of the loop, the maximum number of dependencies is 2, `fib[i-1]` and `fib[i-2]` as described in the inner loop part.

```
fib[0] = 1;
fib[1] = 1;
for(i=2; i<=n; i++)
  fib[i%2] = fib[(i-1)%2] + fib[(i-2)%2];
result = fib[n%2];
```

By transforming the program this way, the correctness of the program is not flawed, notably because the live-out value, `result`, is the same, and we have deduced a storage mapping that reduces the storage requirements; by applying the mapping `fib`$[i] \rightarrow$ `fib`$[i \bmod 2]$, we reduced the storage required from n to 2. This example allowed us to talk about optimizing temporary memory allocation in an hopefully intuitive way. By taking a look at the dependencies between each iteration of the `for` loop, we made ourselves assured that the maximum number of variables that need to be stored at any point in the program was 2.

1.2 Outline

This report is structured as follows. Section 1 uses a preliminary example to motivate the array contraction problem. Section 2 presents the relevant notations we shall use and the framework associated. Section 3 details approaches on this problem from other works. Section 4 presents

our array contraction method, and we discuss its results from its application to various examples in section 4.3.

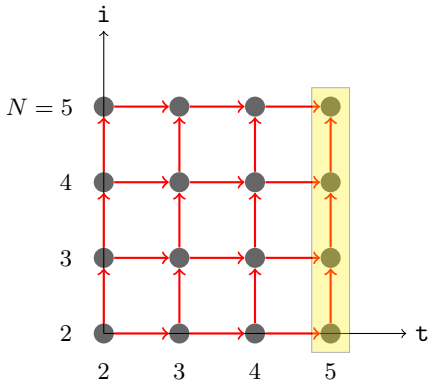
2 Background

We present the necessary background to understand the problem. We define the framework used, namely the *polyhedral model*, the problem at hand, *array contraction*, and the related notions.

We turn to an example named **pc-2D-single**, for *producer-consumer in 2 dimensions, single dependency vector*. Consider the following C-like code, describing the producer-consumer dynamic. `T` represents the main computation, and writes - or *produces* - values to `A` while also reading some values from `A`, and `U` reads - or *consumes* - values from `A`. `S` is the statement representing the left and bottom corner cases where $t = 1$ or $i = 1$ and generates no read dependency; we see the bottom and left corners as inputs.

```
for (t=1; t<=N; t++)
  for (i=1; i<=N; i++)
    if(t==1 || i==1)
      A[t,i] = input();//S
    else
      A[t,i] = A[t,i-1] + A[t-1,i];//T
  for(i=2; i<=N; i++)
    result = result + A[N,i];//U
```

a) Example code



b) Iteration domain of T

Figure 1: pc-2D-single example

We will use this example through the presentation of the theoretical background, and of our contributions. Our focus is on the contraction of the array `A`, which is originally of size $N \times N$. We show that under the specified scheduling (of $\theta_S(t,i) = \theta_T(t,i) = (0,t,i)$ and $\theta_U(i) = (1,i)$), for which we shall introduce the notation in the following section, the size of `A` can be shrinked down to $2 \times (N-1)$.

2.1 Polyhedral model

The *polyhedral model* defined by [7] is an intermediate representation of a loop as a graph over points of $\mathbb{Z}^n$, which gives us a geometrical representation of the statements composing the loop. The class of programs that can be represented in this model, and so are subject to polyhedral optimizations, are named Static Control Parts, (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 $\langle S, \vec{i} \rangle$ where $\overrightarrow{i}$ 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 $\mathbb{Z}^n$.

Definition. *Data dependency.* For two instances $\langle S, \vec{i} \rangle$ and $\langle S, \vec{j} \rangle$, $\langle S, \vec{j} \rangle$ *depends of* $\langle S, \vec{i} \rangle$ (noted $\langle S, \vec{i} \rangle \rightarrow \langle S, \vec{j} \rangle$), if and only if $\langle S, \vec{i} \rangle$ is computed before $\langle S, \vec{j} \rangle$ in the original sequential order (denoted $\langle S, \vec{i} \rangle \prec \langle S, \vec{j} \rangle$) and one writes and the other reads the same array space.

This sequential order $\prec$ is the lexicographical order between instances of the same statement: $\langle S, \vec{i} \rangle \prec \langle S, \vec{j} \rangle$ if and only if $\vec{i} \ll \vec{j}$.

In the computation of the pc-2D example, any instance $\langle S0, t, i \rangle$ depends of its preceding instances $\langle T, t-1, i \rangle$ and $\langle T, t, i-1 \rangle$, coincidentally of iteration vectors $(t-1, i)$ and $(t, i-1)$.

Definition. *Scheduling.* A schedule maps each execution instance $\langle S, \vec{i} \rangle$ to an execution date $\theta_S(\vec{i})$. 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 $\ll$. In a way, a schedule maps each iteration vector to its counterpart in the transformed, *scheduled* program, but it is fundamental to differentiate both. Incidentally, they coincide here in the same way that Figure 1 in section 2 and Figure 2 in section 2.2 in have similar geometrical representations, in that because the iteration domain and the array index domain have the same dimension, and each iteration writes to different array cells.

2.2 Array contraction

The problem of array contraction consists in finding a mapping $A[i] \rightarrow A_{opt}[\sigma(i)]$ reducing the required size of `A`. First, we define the conflict relation for arrays, and then we give a correctness condition for such mappings.

Definition. *Conflict set.* Two array indices $\vec{i}$ and $\vec{j}$ of `A` are in conflict (denoted $\vec{i} \bowtie \vec{j}$) if their lifetime overlap, i.e. if there exists at least one time instant where both array cells access the same memory location, and at least one writes to it.

Formally, we define this conflict as follows, with W_i write of `A[`$\vec{i}$`]`, R_i read of `A[`$\vec{i}$`]`, and so on:

$$\exists W_i, R_i, W_j, R_j \quad s.t. \quad \theta(W_i) \ll \theta(R_i) \quad and \quad \theta(W_j) \ll \theta(R_j) \tag{1}$$

The mapping is *correct* if and only if it satisfies the conflict relation:

$$\vec{i} \bowtie \vec{j}, \ \overrightarrow{i} \neq \overrightarrow{j} \quad \Rightarrow \quad \sigma(\overrightarrow{i}) \neq \sigma(\overrightarrow{j}) \tag{2}$$

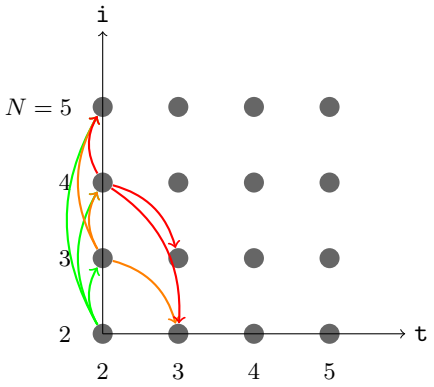


Figure 2: Conflict Set (partial) of pc-2D-single for the 3 first iterations. Points represent array cells.

3 Related work

We quickly go over the multiple works related closely or loosely to our subject. We first present works that served as benchmarks; we then go over closely related work, including those that define the more general context of the polyhedral model; finally, we present loosely related work, that consider the domain of affine loops. As described by [3, 9], 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. The method of Lefebvre and Feautrier [9] gives on the two examples *pc-2D* and *blur-interleaved* storage mappings of $(t \bmod N, i \bmod N)$ and $(y \bmod 3, x \bmod N)$, and the more advanced work of Bhaskaracharya et al. [3] infer more refined mappings, $i - t \bmod (2N - 1)$ and $(2x - y) \bmod (2N + 1)$ respectively, because their approach consider the change to a better basis for the contraction vectors.

The type of optimization we are looking for is an *intra-array optimization* as designed by [3], and references such as [1, 7, 10] focus on this intra-array analysis. [3, 4] give a unified model for intra-array as well as inter-array optimization[1]. There is however a cost as of date; The focus is on Static Control Parts (SCoPs) as formally defined in [2], a class of programs in which loops indices iterate by affine functions of the outside parameters and loop indices. This class of programs can be handled thanks to the Polyhedral model through loop transformations, defined in [7], which allows us to turn source-level loops into intermediate-level representations that are subject to mathematical analysis.

In terms of trace analysis, some work has already covered similar topics such as loop recognition and trace prediction [8] and trace-based affine loop reconstruction [11]. Notably, the latter work manages to reconstruct 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.

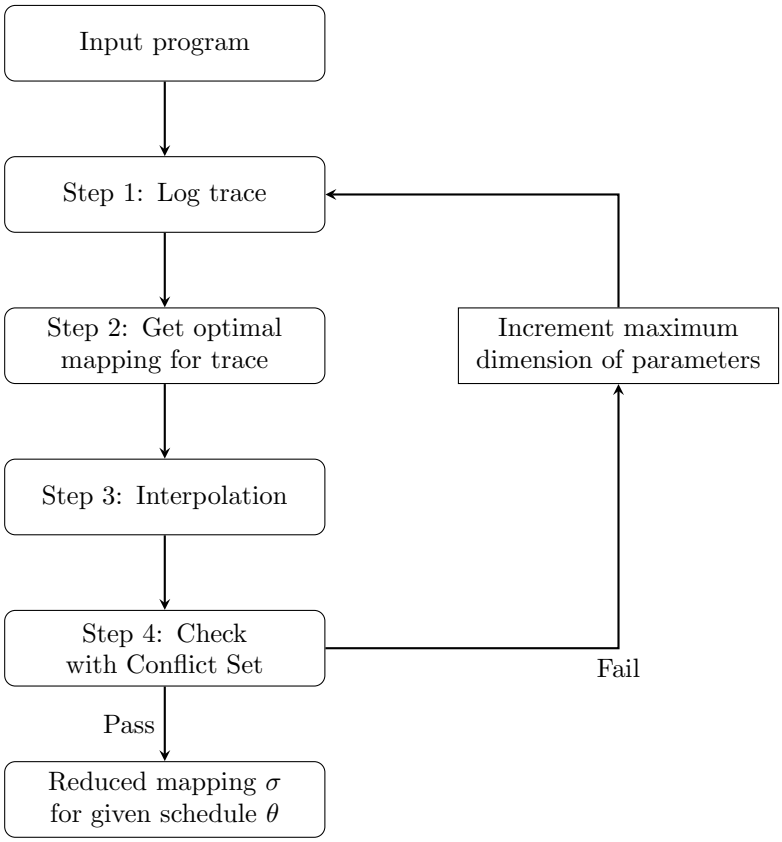
4 Contributions

We present the contributions we made to the problem of array contraction, and detail our method of offline trace execution and analysis to infer a general mapping.

In this report, we introduce an alternative method of creating the Conflict set by performing a liveness analysis on the trace of a given program, and then inferring a reduced correct modulo mapping for a specific array. We implemented a trace generator and a trace analyser for three examples based on benchmarks of the polyhedral community, namely *pc-1D*, *pc-2D* and *blur-interleaved*. We propose two implemented algorithms, respectively to compute the liveness and the maximum width alongside an iteration vector dimension, and show how we can deduce a scalar modulo mapping from those. Finally, we present the use of interpolation to identify the parametrization of the moduli, which have only been scalars up to this point, and we show the resulting modulo mapping and iterations needed to provide a reduced parametrized modulo mapping.

[1] for which the scope of optimizations go beyond each unique array, in a global manner where optimization between different arrays happen.

4.1 Overview of the approach



The approach to get each example up and running was to first design a global method of trace generation and analysis. Starting from the simplest example pc-1D, first generated with *iscc* as other examples, we sought to establish a method to expose that the maximum number of array cells of `A` that are alive at any point of the execution is 2, and since we are in a dynamic context, we decided to implement a simple dataflow liveness algorithm in order to get this information from a specific trace. The second part was manipulating the liveness sets such that we could easily compute a reduced mapping. Finally, the code generation and polyhedral representations, as well as the modulo mapping correctness check were realised by external libraries.

The workflow of our method goes as follows. From an input program, we generate its scheduled C code thanks to the iscc library. Then, we modify the statement definitions such that they may initialize an object that will store the reads and writes done for the surrounding loop parameters.

The information of which statement has emitted such read or write is not used in our analysis. This means that we group up every read and write by their time of execution, and it allows us to lay out the dependencies of each memory cell for each logical time step (because the operations are then ordered sequentially). Therefore, the control points of our programs are said time steps. As mentionned earlier, we will indeed perform a classic dataflow analysis of the *liveness*. For each of them, we want to compute the following sets of array cells:

- $gen(p)$ the set of array cells that are read by p, and therefore become alive
- $kill(p)$ the set of array cells that are written to by p, and therefore become dead
- $In(p)$ the set of array cells that are alive before p
- $Out(p)$ the set of array cells that are alive after p

Because each step p breaks down the program into contiguous parts, we can safely assume that $Out(p) = In(p+1)$, and therefore the two following equations are straightforward to compute: For each step p,

$$In'(p) = (Out(p) \setminus kill(p)) \bigcup gen(p) \tag{3}$$

$$Out'(p) = \bigcup_{s \in succ(p)} In(s) \tag{4}$$

We then perform a backward (time-wise) dataflow analysis of the trace entries in order to create our Conflict Set, and we determine the maximum number of array cells alive at any control point.

4.2 Algorithm

The following algorithm describes the analysis of the trace. This algorithm represents the second step of our workflow, and is subject to be run multiple times for increasing parameters, because the resulted mapping usually does not satisfy the Conflict Set check since it is only composed of scalars, and we therefore need multiple values to interpolate a possible parameter.

Algorithm 1: Find modulo storage mapping m for t the number of trace entries $T \in \overrightarrow{T}$ and $\overrightarrow{A}$ the array

Result: mapping $\overrightarrow{\sigma}$
function GET-MAPPING($\overrightarrow{A}$, $\overrightarrow{T}$)
  $\overrightarrow{In} \leftarrow \emptyset$
  $\overrightarrow{Out} \leftarrow \emptyset$
  for *each trace entry* $\overrightarrow{T}[p]$ *s.t.* $t > p \geq 0$ **do**
    $gen[p] \leftarrow reads(\overrightarrow{T}[p])$
    $kill[p] \leftarrow writes(\overrightarrow{T}[p])$
    $\overrightarrow{In}[p] \leftarrow (Out \backslash kill(p)) \cup gen(p)$
    $Out \leftarrow \overrightarrow{In}(p)$
  end
  $CS \leftarrow \bigcup_p \{(A(\vec{i}), A(\vec{j})) \mid A(\vec{i}), A(\vec{j}) \in In(p)\}$
  $\Delta_{CS} \leftarrow \{\delta = \vec{i} - \vec{j} \mid (A(\vec{i}), A(\vec{j})) \in CS\}$
  for *each array dimension i, starting from 0, in increasing order* **do**
    $m_i \leftarrow 1 + max\{\delta_i \mid (0, ..., 0, \delta_i, ...) \in \Delta_{CS}\}$
  end
  $\hat{\sigma_t} : \vec{i} \rightarrow \vec{i} \bmod \vec{m}$
  $\sigma \leftarrow$ Interpolate $(\hat{\sigma_1}, ..., \hat{\sigma_t})$

An operation is defined as a set of reads and writes done by any statement at a given multi-dimensional logical time.

Let us detail this algorithm, and use the example *pc-2D-single* detailed in. An operation is defined as a set of reads and writes done by any statement at a given multi-dimensional logical time. The information of which statement has emitted such read or write is not used in our analysis. This means that we group up every read and write by their time of execution, and it allows us to lay out the dependencies of each memory cell for each logical time step (because the operations are then ordered sequentially). Therefore, the control points of our programs are said time steps. The usual algorithm for dataflow analysis differs from ours (which is the first loop of Algorithm 1), as it may take multiple iterations to completely propagate the In and Out dependencies created by the *gen* and *kill* of each instance. The goal is to reach a fixed point where $In'(p) = In(p)$ and $Out'(p) = Out(p)$, and since our trace is sequentially ordered and explored backwards (as per the iteration direction of p), one step is sufficient to reach the fixed point.

We also compute the delta set in one inversed pass, which corresponds to the potentially conflicting indices. Then, along each array dimension i, we filter-in the differences δ in ΔIn with non-resolved conflicts: they are prefixed by exactly i zeros, in the same fashion as [9]. Indeed, at dimension 0, the modulo is chosen to satisfy conflicts between `A[`$i_0, ...$`]` and `A[`$i_0^{'}, ...$`]` with $i_0 \neq i_0^{'}$. Hence, it is sufficient to consider the remaining conflicts afterwards (i.e. for the next dimensions) with $i_0 = i_0^{'}$. In general, this essentially boils down to consider δ prefixed by exactly d zeros at dimension d. deduction comes from the observation that if less zeroes are present, there are no conflict to resolve (indices differ, leading to a non-zero width), or if more are present, then the conflict has already been resolved.

We shall then explain the computation of the modulo, first as scalars. The delta having the maximum width in ΔIn represents the greatest number of conflicting array cells in the current difference set. This *max-width* method is what allows us to "count" the maximum number of array cells required per statement execution. However, what we computed is the maximum integer distance in the set, and therefore increment this by 1 so that it corresponds to the number of points that compose the conflict.

Lastly, we take our focus to the identification of parametrized factors in the resulting moduli. The moduli computed up until this point are only scalar, and we therefore need a method to automatically confirm that, for example, pc-2D-single did scale with N in the j-th direction, as its expected modulo mapping is $i, j \rightarrow i \bmod 2, j \bmod N - 1$. Our approach was to use the Lagrange polynomials to interpolate the evolution of each scalar of the modulo mapping. Indeed, we need to interpolate linear functions of parameters for each dimension, and therefore, with several execution traces generated and scalar moduli associated, we try to interpolate a linear function of parameters along each modulo dimension.

Application on pc-2D-single. Let us choose the example of pc-2D-single, and consider the trace generated by the execution with parameter $N = 4$. The trace is as follows:

```
T iteration (4,4) write: A[4,4] read: A[4,3]A[3,4]
liveness: A[3,4] A[4,3]
T iteration (4,3) write: A[4,3] read: A[4,2]A[3,3]
liveness: A[3,3] A[3,4] A[4,2]
T iteration (4,2) write: A[4,2] read: A[4,1]A[3,2]
liveness: A[3,2] A[3,3] A[3,4] A[4,1]
S iteration (4,1) write: A[4,1] read: A[3,1]
liveness: A[3,1] A[3,2] A[3,3] A[3,4]
T iteration (3,4) write: A[3,4] read: A[3,3]A[2,4]
liveness: A[2,4] A[3,1] A[3,2] A[3,3]
T iteration (3,3) write: A[3,3] read: A[3,2]A[2,3]
liveness: A[2,3] A[2,4] A[3,1] A[3,2]
T iteration (3,2) write: A[3,2] read: A[3,1]A[2,2]
liveness: A[2,2] A[2,3] A[2,4] A[3,1]
S iteration (3,1) write: A[3,1] read: A[2,1]
liveness: A[2,1] A[2,2] A[2,3] A[2,4]
S iteration (2,4) write: A[2,4] read: A[2,3]A[1,4]
liveness: A[1,4] A[2,1] A[2,2] A[2,3]
T iteration (2,3) write: A[2,3] read: A[2,2]A[1,3]
liveness: A[1,3] A[1,4] A[2,1] A[2,2]
T iteration (2,2) write: A[2,2] read: A[2,1]A[1,2]
liveness: A[1,2] A[1,3] A[1,4] A[2,1]
S iteration (2,1) write: A[2,1] read: A[1,1]
liveness: A[1,1] A[1,2] A[1,3] A[1,4]
S iteration (1,4) write: A[1,4] read: A[1,3]
liveness: A[1,1] A[1,2] A[1,3]
S iteration (1,3) write: A[1,3] read: A[1,2]
liveness: A[1,1] A[1,2]
S iteration (1,2) write: A[1,2] read: A[1,1]
liveness: A[1,1]
S iteration (1,1) write: A[1,1] read:
liveness:
```

Figure 3: Backwards trace for pc-2D-single; in green, (some) instances of maximum width along $\vec{i}$, in red, the only instances of maximum width along $\vec{j}$

For each entry p of the trace, we compute gen, $kill$, In and Out easily by our simplified liveness equations. We calculate the ΔIn set, for which its largest width t-wise is 1, as we can see in almost any iteration $\gg (1,1)$, and its largest width i-wise is 3, as shown in iteration (4,1). We increment those widths by 1, since we are looking for the number of elements composing the width, not the width itself. We therefore deduce the mapping $\hat{\sigma}_2 : (t,i) \mapsto (t \bmod 2, i \bmod 3)$ to be a potential reduction. The same process with $N = 3$ leads to a mapping $\hat{\sigma}_1 : (t,i) \mapsto (t \bmod 2, i \bmod 2)$, and assuming we computed the trace for $N = 3$ first, then for $N = 4$, we can deduce that the modulo for t scales with the constant 2, and that the modulo for i scales linearly with N, by performing two interpolations, between the points $(N = 3, m_1 = 2)$ and $(N = 4, m_1 = 2)$ to deduce the function $f_t : N \mapsto 2$ and therefore the modulo $t \bmod 2$, and the points $(N = 3, m_2 = 2)$ and $(N = 4, m_2 = 3)$ to deduce the function $f_i : N \mapsto N - 1$ and therefore the modulo $i \bmod N - 1$. Therefore, our method proposes the mapping $\sigma : (t,i) \mapsto (t \bmod 2, i \bmod N - 1)$, which does

respect the conflicts.

4.3 Experiments

4.3.1 Implementation

iscc. *iscc* [2] is a library for manipulating programs of the polyhedral model. It offers tools to automatically generate tiled and scheduled C code from a mathematical description of the iteration domain and schedule. This tool was used to generate the base code of each of the examples, to which we then applied modifications to make the tuned code log instances of the statements. Its direct usage in our implementation corresponds to the "Input program" abstract step.

poco. *poco* is a library that manipulates intermediate representations of SCoPs. From its toolbox, we used its intermediate representation parser from XML to load informations of the program into main memory, let the distantly executed code produce and process its trace to return a possible mapping, then get back and check the mapping against the mapping checker offered by the library. It tries to apply the given mapping and returns the satisfiability of the conflict set of the program for this mapping. Its direct usage correspond to the "Input program" abstract step, alongside the step 3 and the "Increment" conditional.

The examples have been written in C, and our implementation of the trace generation and the algorithms have been written in C++. The implementation of each example features:

- A specialization of each of the example codes, to log its trace (as _trace.cc)
- a Trace.cc file describing the methods to manipulate the trace
- an Arraycontraction.cc file describing the algorithms to analyse the trace and manipulate the sets of indices
- A specialization of the main program for each example, loading the intermediate representation into memory and checking the mapping against the conflict set.

4.3.2 Examples

pc-1D. pc-1D is a very basic example where we have scheduled reads (statement `S1`) on array cells that are written-to two time steps before (statement `S0`). The code that would represent it would go as follows:

```
for(i=1; i<=N; i++)
    A[i] = 0;//S0
for(i=1; i<=N; i++)
  result = A[i];//S1
```

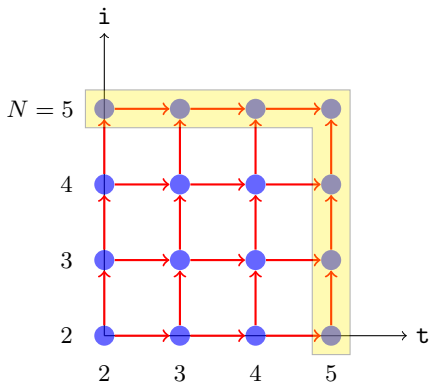
The optimal mapping for this example is $(A[i] \to A[i \bmod 2])$, as no more than two array cells of A are alive at any point of the execution of the program (the first two statement instances each generate one live variable, and each statement instance after that kills one and makes another live, being a simultaneous execution of the two different statements `S0` and `S1`). This example also allows us to show that scheduling plays an important role in the relevance of the possible storage mappings, as here, a scheduling of (i+N) for the second loop would lead to a mod N mapping.

[2] available at http://barvinok.gforge.inria.fr/

pc-2D. pc-2D required a finer analysis than its dimensionally smaller counterpart. Once again, the statement T collects the results of the statement S. The schedule is the sequential order of execution as with all other examples.

```
for(i=1; i<=N; i++)
  for(j=1; j<=N; j++)
    if(t==1 || i==1)
      A[t,i] = input();// S
    else
      A[t,i] = A[t,i-1] + A[t-1,i];// T
for(i=1; i<=N; i++)
  result = result + A[N,i] + A[i,N];//U
```

a) Example code



b) Iteration domain of S

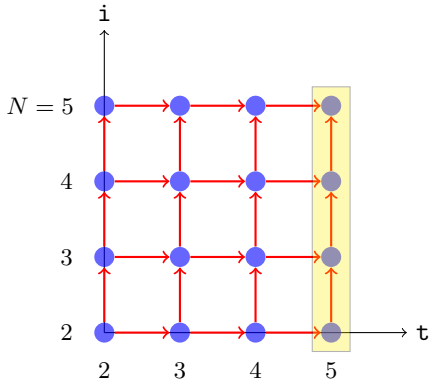
Figure 4: pc-2D example

The modulo mapping to seek here is $(i, j) \rightarrow (i - j \bmod 2N - 1)$, as the entierty of the top and right edges of the polyhedron need to be kept as live-out values, and therefore a minimum of $2N - 1$ array cells are needed for the computations done on `blurx`. Because our method contracts arrays one dimension at a time, our approach results in the same mapping as [7] of $(i, j) \rightarrow (i \bmod N, j \bmod N)$.

pc-2D-single. We decided to study a less complex version of this example, pc-2D-single, where U drops his t-wise dependency vector:

```
for (t=1; t<=N; t++)
  for (i=1; i<=N; i++)
    if(t==1 || i==1)
      A[t,i] = input();//S
    else
      A[t,i] = A[t,i-1] + A[t-1,i];//T
for(i=1; i<=N; i++)
  result = result + A[N,i]; //U
```

a) Example code



b) Iteration domain of T

Figure 5: pc-2D-single example

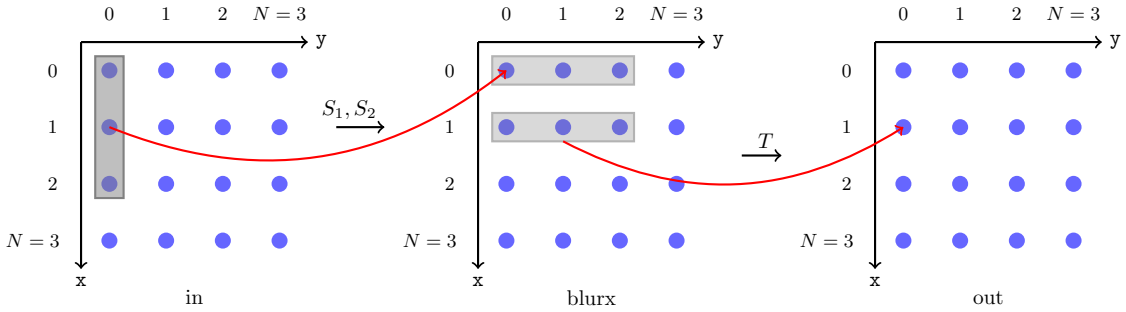
We have inferred that a better mapping on `A` than one of magnitude N^2 would be $(t, i) \rightarrow$

$(t \bmod 2, i \bmod N-1)$, as we are able to remark that the maximum we can trim to along the i-th axis is 2.

blur-interleaved. blur-interleaved is an example of a classic stencil computation where, as the name suggests, the consumer instances are interleaved with the producer ones, making pen-and-paper analysis a more difficult process.

```
for(y=0; y<2; y++)
  for(x=0; x<N; x++)
    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++) {
    blurx[x,y] = in[x,y] + in[x+1,y] + in[x+2,y];
    out[x,y] = blurx[x,y-2] + blurx[x,y-1] + blurx[x,y];
}
```

a) Example code



b) Dependencies for in, blurx and out through S_1, S_2 and T

Figure 6: blur-interleaved example

This example shows the use of two successive convolutions, one on `in`, as input to the temporary array `blurx`, and one from `blurx` to `out`. The instances of the write and read accesses to `blurx` are interleaved as soon as there is enough to run the first instance of the statement `T`. We find the same result as Lefebvre and Feautrier's (listed as a baseline in [3]) on the array `blurx`, the mapping being $(y, x) \to (y \bmod 3, x \bmod N)$.

5 Preliminary results

In this section, we present our work as the implementation of the trace generation, analysis, and inference of the optimal mapping function for three examples: producer-consumer (in 1D and 2D) and blur-interleaved.

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 made by an external polyhedral library named Poco, created by Christophe Alias, 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 a offline execution trace, generated by 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 C kernels have been compiled using gcc 9.3.0, while the implementation itself has been compiled using g++ 9.3.0. Every compilation and execution of the kernels, and therefore the time measurements on them have been done on an Intel Core i5-1135G7 CPU running at 2.40GHz. The benchmarks are mostly inspired from the ones of the polyhedral community, as pc-2d and blur-interleaved are 2D stencils, whereas pc-1d is a very basic 1D stencil.

5.2 Results

In this table, the base mapping represents the original size of the array, and the mapping deduced is the reduced size inferred from our algorithm. Iteration count represents the number of executions of the online trace, for the associated parameters specified. The average runtime describes the measured time spent on the inference algorithm execution. We can observe that the time cost is an order of magnitude higher for 2D stencils compared to pc-1D, which leads us to believe that improvement is possible. As we have yet to compare as of now with other work, we cannot clearly size the efficacy of our method for now. However, we can still remark that runtimes are in the tens of milliseconds for popular kernels such as blur-interleaved. We support therefore the idea that dynamic analysis on smaller inputs are way less costly that full-on static analysis, the latter fully relying on polyhedral intermediate representations and operations, and Integer Linear Programming.

Kernel	Array	Base mapping	Mapping deduced	Iteration count	Avg. runtime (ms)
pc-1D	`A`	$i \bmod N$	$i \bmod 2$	2 (for N=2,3)	1.04
pc-2D	`A`	$i \bmod N, j \bmod N$	$i \bmod N, j \bmod N$	2 (for N=2,3)	13.0
pc-2D-single	`A`	$i \bmod N, j \bmod N$	$i \bmod 2, j \bmod N - 1$	2 (for N=2,3)	14.2
blur-interleaved	`blurx`	$i \bmod N, j \bmod N$	$i \bmod 3, j \bmod N$	2 (for N=3,4)	13.1

Table 1: Reduced storage mappings proposed for given example kernels

6 Conclusion and future work

We finally conclude by pointing out that our results match our expectations, and open on the future of our research.

In this paper, we have taken a first step to explore what dynamic analysis can bring to the problem of temporary memory allocation. We have presented a method to compute the exact liveness inside of a SCoP, and a method to infer a modulo mapping given a canonical basis. We have also shown that interpolation of the parameters from a subset of the possible traces is possible, and we have applied it to our examples. We have shown results that match the magnitude of those of other known work [1, 3], and answered positively to the question of generalization from a subset of traces.

In the future, we seek to apply the same process to more benchmarks of the polyhedral community, as our experimentations were limited by time. We also look forward to apply finer analysis on the modulo parametrization, like heuristics to choose parameter values that are relevant to both the conflict set checking and the interpolation (given that for values too low, the iteration domain can be oddly defined), and more generally a finer approach to the choice of basis for any array allocation problem.

References

[1] Christophe Alias, Fabrice Baray, and Alain Darte. 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] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 209–225. Springer, 2003.

[3] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(3):1–23, 2016.

[4] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 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*, pages 526–538, 2016.

[5] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[6] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, 2005.

[7] Paul Feautrier and Christian Lengauer. Polyhedron model. *Encyclopedia of parallel computing*, 1:1581–1592, 2011.

[8] Alain Ketterlin and Philippe Clauss. 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*, pages 94–103, 2008.

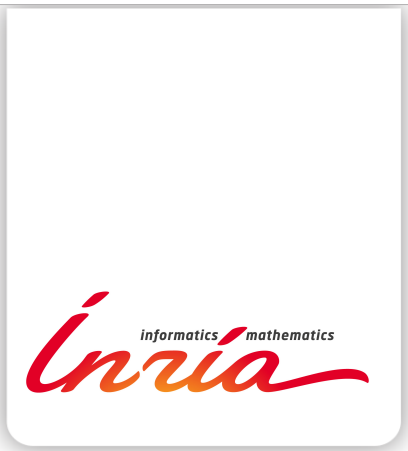
[9] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel computing*, 24(3-4):649–671, 1998.

[10] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[11] Gabriel Rodríguez, José M Andión, Mahmut T Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 139–149, 2016.

[12] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. Polyjit: Polyhedral optimization just in time. *International Journal of Parallel Programming*, 47(5):874–906, 2019.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399