HAL
open science

Lattice-Based Array Contraction: From Theory to Practice

Christophe Alias, Alain Darte, Fabrice Baray

To cite this version:

Christophe Alias, Alain Darte, Fabrice Baray. Lattice-Based Array Contraction: From Theory to Practice. [Research Report] LIP RR-2007-44, LIP - Laboratoire de l'Informatique du Parallélisme. 2007. hal-02127064

HAL Id: hal-02127064

https://hal-lara.archives-ouvertes.fr/hal-02127064v1

Submitted on 13 May 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Lattice-Based Array Contraction: From Theory to Practice

Christophe Alias
Fabrice Baray
Alain Darte

November 2007

Research Report N° 2007-44

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Lattice-Based Array Contraction: From Theory to Practice

Christophe Alias, Fabrice Baray, and Alain Darte

November 2007

Abstract

We build on prior work on intra-array memory reuse, for which a general theoretical framework was proposed based on lattice theory. Intra-array memory reuse is a way of reducing the size of a temporary array by folding, thanks to affine mappings and modulo operations, reusing memory locations when they contain a value not used later. We describe the algorithms needed to implement such a strategy. Our implementation has two parts. The first part, Bee, uses the source-to-source transformer ROSE to extract from the program all necessary information on the lifetime of array elements and to generate the code after memory reduction. The second part is a stand-alone mathematical tool dedicated to optimizations on polyhedra, in particular the computation of successive minima and the computation of good admissible lattices, which are the basis for lattice-based memory reuse. Both tools are developed in C++ and use linear programming and polyhedra manipulations. These tools can be used either for memory reduction in programs – e.g, to limit memory expansion introduced for parallelization – or for memory reduction in high-level synthesis – e.g., to design adequate memories between communicating hardware accelerators.

Keywords: Memory reduction, program analysis and source transformations, integer lattices

Résumé

Nous étendons les résultats (théoriques) antérieurs concernant la réutilisation mémoire basée sur les réseaux entiers. Celle-ci permet de réduire la taille des tableaux temporaires, grâce à des fonction d'accès affines et des modulos, en réutilisant des cases contenant des valeurs non lues plus tard. Nous décrivons les algorithmes nécessaires pour mettre en oeuvre une telle stratégie. La première partie de notre implantation, Bee, utilise la bibliothèque de transformation au niveau source, ROSE, pour extraire du programme les durées de vie des cases des tableaux et pour générer le code après réduction mémoire. La seconde, Cl@k, est un programme indépendant d'optimisations sur les polyhèdres, effectuant en particulier le calcul des minima successifs et d'un bon réseau entier admissible, bases de cette réutilisation mémoire. Les deux programmes ont été développés en C++ et utilisent la programmation linéaire et les manipulations de polyhèdres. On peut s'en servir soit en optimisation de programmes, par exemple pour limiter l'expansion mémoire introduite par la parallélisation, soit en synthèse de haut niveau, par exemple pour concevoir des mémoires adaptées entre des accélérateurs matériels communiquants.

Mots-clés: Réduction mémoire, transformation et analyse de programmes, réseaux entiers

Lattice-Based Array Contraction: From Theory to Practice

Christophe Alias, Fabrice Baray, and Alain Darte

19th November 2007

Abstract

We build on prior work on intra-array memory reuse, for which a general theoretical framework was proposed based on lattice theory. Intra-array memory reuse is a way of reducing the size of a temporary array by folding, thanks to affine mappings and modulo operations, reusing memory locations when they contain a value not used later. We describe the algorithms needed to implement such a strategy. Our implementation has two parts. The first part, Bee, uses the source-to-source transformer ROSE to extract from the program all necessary information on the lifetime of array elements and to generate the code after memory reduction. The second part is a stand-alone mathematical tool dedicated to optimizations on polyhedra, in particular the computation of successive minima and the computation of good admissible lattices, which are the basis for lattice-based memory reuse. Both tools are developed in C++ and use linear programming and polyhedra manipulations. These tools can be used either for memory reduction in programs – e.g, to limit memory expansion introduced for parallelization – or for memory reduction in high-level synthesis – e.g., to design adequate memories between communicating hardware accelerators.

1 Introduction

The optimization of memory in multimedia applications for embedded systems has received a lot of attention in the past years, for reducing both memory transfers and memory storage, which have a strong impact on power consumption, performance, and chip area. In this paper, we focus on memory storage reduction, which is important from both an architecture and an application point of view.

On the architecture side, an important characteristic of embedded systems is that the hardware, in particular memories, can be customized. When designing and optimizing a programmable embedded system, the designer wants to select the adequate parameters for cache and scratch-pad memories to achieve the smallest cost for the right performance for a given application or set of applications. In high-level synthesis, when designing non-programmable hardware accelerators, the designer or synthesizer can even fully design the memories (size, topology, connections between processing elements) and customize them for a given application. Embedded systems are thus good targets for memory optimizations. High-level synthesis projects such as [12, 26] need to rely on powerful compile-time

program and memory analysis to be able to automatically or semi-automatically generate a fully-customized circuit from a high-level C-like description. For programmable embedded systems, run-time memory reduction techniques are also possible (see for example [27]), but it is not the topic of this paper.

On the application side, multi-media applications often make intensive use of multi-dimensional arrays, in sequences of loops, or even sequences of nested loops, which make them good target for static program analysis. Before the final implementations, the applications need to be rewritten several times, either by the compiler or the developer, to go from a high-level algorithmic description down to an optimized and customized version. For memory optimizations, the high-level description is where the largest gain can be obtained because global program analysis and global code transformations can be done. For this, it is therefore important to analyze multi-media applications at the source level.

We present an implementation of intra-array memory reuse based on lattice theory in the source-to-source transformer ROSE [19], developed at the Lawrence Livermore National Labs. Intra-array memory reuse was first proposed by De Greef, Catthoor, and De Man [7] at IMEC, as a mean to reduce the size of temporary arrays. Basically, their technique was to linearize a temporary array in some canonical way and to fold it with a modulo operation, thus reducing its size, so that a memory cell can be reused when it contains a dead value, i.e., a value no longer used. This problem was then considered by several authors, with different viewpoints and techniques [13, 18, 23, 24], including some more work at IMEC [25], until a more general framework was proposed in [5] that tries to integrate all approaches into a unique setting. We build on this theoretical work and we bridge the gap between its formal description in [5, 6] and the algorithmic needs, both for program analysis and memory reduction, to really use such a framework in a compiler.

Related work Memory reduction has given rise to a large amount of work recently. In addition to intra-array memory reuse already mentioned previously, there are two other important aspects in memory reduction: memory size estimation and loop transformations for memory reuse.

The first aspect is to be able to give estimations on the number of values live at a given time [30, 31, 1, 28, 2]. Such estimations can be lower bounds or upper bounds. Lower bounds give indications and guides for further loop transformations, while upper bounds can also be used for memory reduction. Although some of these techniques can give parametric memory estimations, i.e., closed-form expressions in the parameters of the program, they are of no use for building an appropriate memory mapping that can actually lead to such a memory size. This is the main difference with intra-array memory reuse, which leads to actual memory allocations. Note also that the memory allocations that we derive are, by nature, upper bounds for memory size estimation.

The second aspect of memory reduction is to go beyond memory optimization of an already-scheduled program and to transform the program, i.e., to schedule it in another way, so that it consumes less memory. Typically, the goal is to reduce the lifetime of temporary values and increase the temporal reuse of values. Array contraction (see for

example [22, 4]) is a form of memory reduction that can be achieved with program transformation, in particular loop fusion. All the classical work on loop transformations for data locality, starting from [29], are also relevant here, even if they do not exactly target memory reduction. We cannot cite them all. An interesting recent work is [20], which studies the effect of loop transformations on lifetimes of array variables.

2 Background Notations and Definitions

We use the classical notations used for describing, analyzing, and transforming codes with nested loops and multi-dimensional arrays, as introduced in [9] as static control programs. Each particular execution of a statement S – an *operation* – is represented by $(S, \vec{i})$ where $\vec{i}$ is the *iteration vector*, which represents the loop indices of the surrounding loops. We make the standard assumption that loops are *affine*, i.e., a loop index takes all possible integer values from the loop lower bound to the loop upper bound and these bounds are max (resp. min) of affine expressions of surrounding indices. In other words, for each statement S, the iteration vector $\vec{i}$ spans, in lexicographic order, all integer points of an iteration domain $\mathcal{I}_S$, which is a polyhedron. This affine framework also accepts if-statements as long as conditions can be analyzed as affine inequalities on loop indices. Similarly, we assume that arrays are accessed with affine functions of loop indices, otherwise some conservative approximations need to be done, when analyzing the program, as done in [17, 3].

For memory reduction, we focus on *temporary arrays*, i.e., those used for intermediate computations in a procedure and that are neither live-in, nor live-out. Only intra-procedural information is needed to optimize them. We want to derive a storage-saving mapping $\sigma(\vec{i})$ from the index space of a given array A to a set of storage locations indexed by addresses, with $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$, where M is a matrix, $\vec{b}$ is a vector, and the modulo operation is applied component-wise. Such a mapping σ is called a *modular mapping* and denoted by $(M, \vec{b})$. The smallest the product of the components of $\vec{b}$, the smallest the required memory. To get an adequate matrix M and an adequate vector $\vec{b}$, our goal is to make practical the formalism and techniques proposed by Darte, Schreiber, and Villard in [5, 6]. This formalism generalizes, at least in theory, the previous techniques proposed, in particular, by De Greef *et al.* [7], Lefebvre and Feautrier [13], Quilleré and Rajopadhye [18], Strout *et al.* [23], Thies *et al.* [24].

Figure 1 depicts the main steps of our array contraction method. These steps are described below.

Step 1 For deriving memory allocations with intra-array reuse, a possibility is to first compute, in an abstract way, for each array index, the first time it is written (first write) and the last time it is read (last read). We will make sure that, between the first write and the last read, the memory location is not overwritten when using the modular mapping σ: we call this "interval" the lifetime of the array index. As pointed out in [6], when a given array location is written several times, we could be more accurate, distinguish the

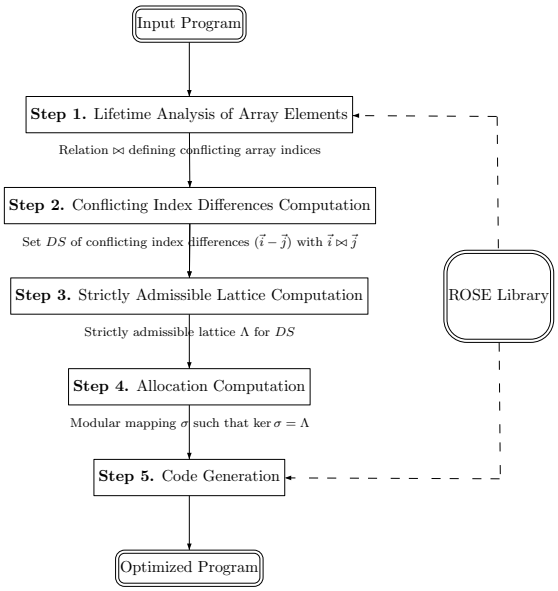


Figure 1: Overview of the method

different defined values, and work with a union of "intervals" (between each write and its last corresponding read), instead of a single one. But this would make everything more complicated. Also, a pre-transformation of the code, renaming arrays, is possible if needed to get the same effect. We do not address this extension in our current implementation.

Step 2 Once the lifetime of array indices is analyzed, we can define conflicting indices: two array indices $\vec{i}$ and $\vec{j}$ conflict (denoted by $\vec{i} \bowtie \vec{j}$) if their lifetimes intersect. We will make sure that such indices are not mapped to the same memory location through σ, i.e., $\vec{i} \bowtie \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$. If DS denotes the set of all $\vec{d}$ such that $\vec{d} = \vec{i} - \vec{j}$, with $\vec{i} \bowtie \vec{j}$, the previous condition is equivalent to $\vec{d} \in DS, \sigma(\vec{d}) = 0 \Rightarrow \vec{d} = 0$, i.e., $\ker(\sigma) \cap DS = \{0\}$ where $\ker(\sigma)$ is the set of all $\vec{i}$ such that $\sigma(\vec{i}) = 0$ (the kernel of σ). Step 2 consists in building this set DS of conflicting index differences or an over-approximation. Steps 1 and 2 are detailed in Section 3.

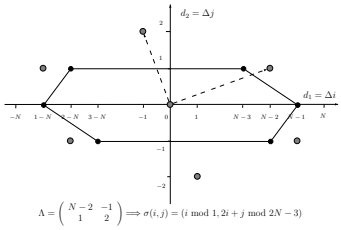
Step 3 The kernel of a modular mapping $\sigma = (M, \vec{b})$ from $\mathbb{Z}^n$ to $\mathbb{Z}^p$ is a sublattice of $\mathbb{Z}^n$. A lattice Λ with $\Lambda \cap DS = \{0\}$ is a *strictly admissible lattice* for DS. Given such an integer lattice, it is easy to build a mapping $(M, \vec{b})$ whose kernel is Λ and such that the product of the components of $\vec{b}$ (the memory size) is equal to the determinant of Λ (Step 4). In other words, the search for a good modular mapping is reduced to the problem of determining a strictly admissible integer lattice for DS with small determinant. Algorithms and heuristics to find such a lattice are detailed in Section 4.

Steps 4 and 5 are straightforward and will not be detailed. Step 4 consists in building, from a lattice Λ, a matrix M and a vector $\vec{b}$ so that the kernel of $(M, \vec{b})$ is Λ. This may need some standard matrix computations (Hermite or Smith normal forms). Heuristics

```
  #define N 100

S1 y[0][0] = r[0];
   beta[0] = 1;
   alpha[0] = r[0];
   for(k=1; k<=N-1; k++) {
     beta[k]=
       beta[k-1] - beta[k-1]*(alpha[k-1])^2
     sum[0][k]=r[k];
     for(i=0; i<=k-1; i++)
S2     sum[i+1][k] =
         sum[i][k] + r[k-i-1]*y[i][k-1];
     alpha[k] = -sum[k][k] * beta[k];
     for(i=0; i<=k-1; i++)
S3     y[i][k] =
         y[i][k-1] + alpha[k]*y[k-i-1][k-1];
S4   y[k][k] = alpha[k];
   }
   for(i=0; i<=N-1; i++)
S5   out[i] = y[i][N-1];
```

(a)



(b)

```
  #define N 100
  #define M 197

S1 y[0] = r[0];
   beta = 1;
   alpha = r[0];
   for(k=1; k<=N-1; k++) {
     beta = beta - beta*(alpha)^2
     sum = r[k];
     for(i=0; i<=k-1; i++)
S2     sum = sum + r[k-i-1]*y[(2*i+k-1)%M];
     alpha = -sum * beta;
     for(i=0; i<=k-1; i++)
S3     y[(2*i+k)%M] = y[(2*i+(k-1))%M] +
                alpha*y[(3*k-2*i-3)%M];
S4   y[(3*k)%M] = alpha;
   }
   for(i=0; i<=N-1; i++)
S5   out[i] = y[(2*i+N-1)%M];
```

(c)

Figure 2: (a) Durbin's kernel, (b) Set DS of conflicting index differences for array y (polytope), an optimal integer lattice Λ (grey points), and a corresponding mapping σ (all other temporary arrays are scalarizable), (c) Transformed program.

developed in Step 3 can also give directly the modular mapping. See [6] for more details, we just use what is explained there. Finally, Step 5 consists in replacing each access $A(f(\vec{i}))$ to an optimized array A by the reference $A(\sigma(f(\vec{i})))$. This, again, is done thanks to the source-to-source transformer ROSE [19]. Some refinements on σ can be done, which are mentioned in Sections 3 and 4.

Main example We will use, as running example, the classical Durbin's kernel, which solves a Toeplitz system with N unknowns. See its code in Figure 2.(a). The array `r` is an input array, the array `out` is an output array, the arrays `alpha`, `beta`, `sum`, and `y` are temporary arrays, thus subject to memory reduction. In this example, any sophisticated tool should detect that arrays `alpha`, `beta`, and `sum` can be replaced by scalars. For array `sum`, this check requires inter-loop analysis. Our tool finds easily these complete memory reductions because, for each of these arrays, the set of conflicting index differences DS is reduced to $\{0\}$. More interesting is the case of array `y` that can be optimized but not completely. It is not too difficult to check that `y` can be folded into an array of size $N \times 2$, with the mapping $\sigma(i_1, i_2) = (i_1, i_2 \bmod 2)$. In other words, it is correct to replace, in the code, all occurrences `y`$[c_1][c_2]$ by `y`$[c_1][c_2 \bmod 2]$. We will illustrate how to use DS to find automatically such a mapping.

We point out that the best modular mapping for `y` is $\sigma(i_1, i_2) = 2i_1 + i_2 \bmod (2N-3)$, with a slight memory size improvement compared to the previous mapping. Being able to derive automatically such a mapping, in a parametric way and for a general situation, is, to our knowledge, an open problem. However, for a fixed and small enough value of N, our implementation of the exhaustive search proposed in [6] can find such a mapping. For this particular example, the gain of 3 memory cells is certainly not worth the price of the complicated modulo expression. However, for the purpose of illustration, we give in Figure 2.(b) and (c) the set DS and the transformed program with contracted arrays. ■

3 Lifetime Analysis of Array Elements

This section describes the algorithms, used in the first part of our tool, to compute the set DS of conflicting index differences, needed to compute modular allocations. This amounts to compute the conflict relation $\bowtie$, which is done thanks to an accurate lifetime analysis on array elements. We illustrate this analysis with the array y of our main example.

3.1 Exact Lifetime Analysis

The *lifetime* of an array cell $\vec{c}$ is the "time interval" between its initialization and its last read. Classical methods handle arrays as scalar variables and make a too rough approximation of control flow to be used here. This section explains how to compute the first operation writing $\vec{c}$ (its initialization) and the last operation reading $\vec{c}$, for static control programs. Our method is similar on many aspects to exact instance-wise dataflow analysis and revisit some ideas of [9].

3.1.1 First Write of an Array Element

Consider a statement S in a static control program:

$$S: \quad A[u(\vec{i})] = \ldots$$

where $\vec{i}$ is the iteration vector of S and u an affine function. Consider a cell $\vec{c}$ of A and denote by $\mathbf{W}_S(\vec{c})$ the set of operations $(S, \vec{i})$ writing $A[\vec{c}]$. Each operation must be valid and write $A[\vec{c}]$, i.e., $\mathbf{W}_S(\vec{c}) = \{(S, \vec{i}) \mid \vec{i} \in \mathcal{I}_S,\ u(\vec{i}) = \vec{c}\}$. Writing $S_1, \ldots, S_n$ the statements assigning A and denoting by $\preceq$ the execution order between operations, the first operation writing $A[\vec{c}]$ is $\mathbf{FW}(\vec{c}) = \min_{\preceq} \bigcup_i \mathbf{W}_{S_i}(\vec{c})$, rewritten as:

$$\mathbf{FW}(\vec{c}) = \min_{\preceq}\{\min_{\preceq} \mathbf{W}_{S_1}(\vec{c}), \ldots, \min_{\preceq} \mathbf{W}_{S_n}(\vec{c})\} \tag{1}$$

Following [9], the sequencing predicate between two operations $(S, \vec{i})$ and $(T, \vec{j})$ can be defined thanks to the strict lexicographic order $<_l$ between iteration vectors: $(S, \vec{i}) \prec (T, \vec{j})$ iff a) $\vec{i}[1..M] <_l \vec{j}[1..M]$ or b) $\vec{i}[1..M] = \vec{j}[1..M]$ and S is before T in the text order, where M is the depth of the maximum common loop nest of S and T. In particular, $\min_{\preceq} \mathbf{W}_S(\vec{c})$ boils down to compute the lexicographic minimum of the polyhedron $\{\vec{i} \mid \vec{i} \in \mathcal{I}_S,\ u(\vec{i}) = \vec{c}\}$. When the polyhedron depends on parameters (as $\vec{c}$ here), classical algorithms of integer linear programming cannot be applied and we need to use parametric integer programming techniques [8]. The tool PIP [16] implements such techniques and outputs a selection tree, called QUAST (QUasi-Affine Selection Tree), whose nodes are affine relations on parameters and whose leaves provide the relevant lexicographic minimum for each subset defined by these relations.

Main example (cont'd) By definition, the set $\mathbf{W}_{S3}(\vec{c})$ of instances of S_3 writing y$[c_1][c_2]$, with $\vec{c} = (c_1, c_2)$, is:

$$\{(S_3, k, i) \mid 1 \leq k \leq N-1,\ 0 \leq i \leq k-1,\ i = c_1,\ k = c_2\}$$

which is reduced to the operation (S_3, c_2, c_1), whenever (c_2, c_1) is a valid point of the iteration domain of S_3. In general, we use PIP to find the lexicographic minimum. Here, since the access function u is one-to-one, we can just use the iteration domain inequalities and get the QUAST:

$$\min_{\preceq} \mathbf{W}_{S3}(c_1, c_2) = \begin{array}{l} \underline{\textbf{if}}\ N \geq 2\ \underline{\textbf{then}} \\ \quad \underline{\textbf{if}}\ 1 \leq c_2 \leq N-1\ \underline{\textbf{then}} \\ \quad\quad \underline{\textbf{if}}\ 0 \leq c_1 \leq c_2 - 1\ \underline{\textbf{then}} \\ \quad\quad\quad (S_3, c_2, c_1) \\ \quad\quad \underline{\textbf{else}}\ \perp \\ \quad \underline{\textbf{else}}\ \perp \\ \underline{\textbf{else}}\ \perp \end{array}$$

where $\perp$ stands for the empty solution. ■

Once the different $\min_{\preceq} \mathbf{W}_{S_i}(\vec{c})$ are computed, it remains to combine them in order to find the global first $\mathbf{FW}(\vec{c})$ as precised in Equation 1. Since the $\min_{\preceq} \mathbf{W}_{S_i}(\vec{c})$ are QUASTs of operations, they cannot be compared directly with the sequencing predicate. In [9], Feautrier provides simple rules to get the $\min_{\preceq}\{Q_1, Q_2\}$ given two QUASTs Q_1 and Q_2:

$$\begin{array}{l} \min_{\preceq}\{Q, \perp\} \xrightarrow{1} Q \\ \min_{\preceq}\{(S, \vec{i}), (T, \vec{j})\} \xrightarrow{2} \underline{\textbf{if}}\ (S, \vec{i}) \preceq (T, \vec{j})\ \underline{\textbf{then}}\ (S, \vec{i}) \\ \qquad\qquad\qquad\qquad\qquad\qquad \underline{\textbf{else}}\ (T, \vec{j}) \\ \min_{\preceq}\{Q, \underline{\textbf{if}}\ C\ \underline{\textbf{then}}\ Q_1\ \underline{\textbf{else}}\ Q_2\} \xrightarrow{3} \underline{\textbf{if}}\ C\ \underline{\textbf{then}}\ \min_{\preceq}\{Q, Q_1\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \underline{\textbf{else}}\ \min_{\preceq}\{Q, Q_2\} \end{array}$$

Symmetric cases need to be considered too, as in [9].

Main example (cont'd) Due to space limitations, we do not provide the full QUAST of $\mathbf{FW}(\vec{c})$. Instead, we show an equivalent set of clauses $(\vec{i} \in D_k : \omega_k(\vec{i}))_{k \in [\![1,n]\!]}$ where the D_k are disjoint polyhedra and the $\omega_k(\vec{i})$ are operations. Such a set of clauses will be referred as a *flattened QUAST*. We get:

$$\mathbf{FW}(c_1, c_2) = \begin{cases} c_1 = 0,\ c_2 = 0 & (S_1,) \\ \\ 0 \leq c_2 \leq N-1 & \\ 0 \leq c_1 \leq N-2 & (S_3, c_2, c_1) \\ c_1 < c_2 & \\ \\ 0 \leq c_1 \leq N-1 & \\ 0 \leq c_2 \leq N-1 & (S_4, c_1) \\ c_1 = c_2 & \end{cases}$$

Here, the computation of the local first writes $\min_{\preceq} \mathbf{W}_{S_i}(\vec{c})$ is easy as `y` is in single assignment. The main difficulty is the combination of QUASTs to get the global first write. One could also try to remove redundant constraints. ■

3.1.2 Last Read of an Array Element

In the same manner than for the first write, the last operation reading an array cell is derived from local reading sets. To make explanations simpler, each statement is assumed to contain at most one reference to array A. Given a statement S reading $A[u(\vec{i})]$, we let $\mathbf{R}_S(\vec{c}) = \{(S, \vec{i}) \mid \vec{i} \in \mathcal{I}_S,\ u(\vec{i}) = \vec{c}\}$. Writing $S_1, \ldots, S_n$ the statements reading A, the last operation reading $A[\vec{c}]$ is $\mathbf{LR}(\vec{c}) = \max_{\preceq} \bigcup_i \mathbf{R}_{S_i}(\vec{c})$, rewritten as:

$$\mathbf{LR}(\vec{c}) = \max_{\preceq}\{\max_{\preceq} \mathbf{R}_{S_1}(\vec{c}), \ldots, \max_{\preceq} \mathbf{R}_{S_n}(\vec{c})\}$$

We first derive each local last read $\max_{\preceq} \mathbf{R}_{S_i}(\vec{c})$, which boils down to compute the lexicographic maximum of the parameterized polyhedron $\{\vec{i} \mid \vec{i} \in \mathcal{I}_{S_i},\ u(\vec{i}) = \vec{c}\}$. Then, the resulting QUASTs are combined to obtain the global $\max_{\preceq}$, using the same rules as before, where each min is replaced by a max and the right hand-side of rule 2 by **<u>if</u>** $(S, \vec{i}) \preceq (T, \vec{j})$ **<u>then</u>** $(T, \vec{j})$ **<u>else</u>** $(S, \vec{i})$.

Main example (cont'd) Our algorithm finds automatically the following last read of $\mathrm{y}[c_1][c_2]$:

$$\mathbf{LR}(c_1, c_2) = \begin{cases} \begin{array}{l} 0 \le c_1 \le N-2 \\ 0 \le c_2 \le N-2 \\ 2c_1 > c_2 \\ c_1 \le c_2 \end{array} & (S_3, c_2+1, c_1) \\ \\ \begin{array}{l} 0 \le c_1 \le N-2 \\ 0 \le c_2 \le N-2 \\ 2c_1 \le c_2 \end{array} & (S_3, c_2+1, c_2-c_1) \\ \\ \begin{array}{l} 0 \le c_1 \le N-1 \\ c_2 = N-1 \end{array} & (S_5, c_1) \end{cases}$$

Let us explain this result intuitively. First, note that, for each $(k, i) \in \mathcal{I}_{S_2}$, the read of $\mathrm{y}[i][k-1]$ in (S_2, k, i) is then repeated in (S_3, k, i). Thus, none of the instances of S_2 can be a last read for `y`. Now, let us focus on S_3. For each $(k, i) \in \mathcal{I}_{S_3}$, $k < N-1$, the array cell $\mathrm{y}[i][k]$ written by (S_3, k, i) is read twice, by $(S_3, k+1, i)$ (for the reference `y[i][k-1]`) and $(S_3, k+1, k-i)$ (for the reference `y[k-i-1][k-1]`). The last read of $\mathrm{y}[i][k]$ is thus $\max_{\preceq}\{(S_3, k+1, i), (S_3, k+1, k-i)\}$, which boils down to find the maximum of i and $k-i$, with $i < k$, i.e., i when $2i > k$ (first clause) and $k-i$ when $2i \le k$ (second clause). Finally, with $(k, i) \in \mathcal{I}_{S_3}$, $k = N-1$, the array cell $\mathrm{y}[i][k]$ will be read in (S_5, i) (last clause). ■

3.2 Computing the Conflict Relation

As explained in Section 2, two array indices $\vec{i}$ and $\vec{j}$ conflict if their lifetimes $[\mathbf{FW}(\vec{i}), \mathbf{LR}(\vec{i})]$ and $[\mathbf{FW}(\vec{j}), \mathbf{LR}(\vec{j})]$ intersect. In other words, the conflict relation $\bowtie$ is:

$$\vec{i} \bowtie \vec{j} \iff \mathbf{FW}(\vec{i}) \prec \mathbf{LR}(\vec{j}) \text{ and } \mathbf{FW}(\vec{j}) \prec \mathbf{LR}(\vec{i})$$

With the quasts $\mathbf{FW}(\vec{i}) = (\vec{i} \in D_k : \omega_k(\vec{i}))_{k \in [\![1,p]\!]}$ and $\mathbf{LR}(\vec{i}) = (\vec{i} \in D'_k : \omega'_k(\vec{i}))_{k \in [\![1,q]\!]}$ which define partitions of the index set, we get the relation $\bowtie$ through a union of sets. We consider all p^2q^2 sets $C_{k,l,m,n}$ (many are empty) defined for $k \in [\![1,p]\!]$, $l \in [\![1,q]\!]$, $m \in [\![1,p]\!]$, $n \in [\![1,q]\!]$, by:

$$C_{k,l,m,n} = \{(\vec{i},\vec{j}) \mid \vec{i} \in D_k \cap D'_n,\ \vec{j} \in D'_l \cap D_m,\ \omega_k(\vec{i}) \prec \omega'_l(\vec{j}),\ \omega_m(\vec{j}) \prec \omega'_n(\vec{i})\}$$

Then

$$\vec{i} \bowtie \vec{j} \Leftrightarrow (\vec{i},\vec{j}) \in \bigcup_{k=1}^{p} \bigcup_{l=1}^{q} \bigcup_{m=1}^{p} \bigcup_{n=1}^{q} C_{k,l,m,n}$$

Decomposing the predicate $\prec$ into affine inequalities, we can write $C_{k,l,m,n}$ itself a union of sets, and more precisely of polytopes. One can show that p and q are always finite non-parametric values, of the same order than the number P of statements of the program. This consequently provides an $O(P^4)$ algorithm to compute the conflict relation. Despite its apparent complexity, this method is fast enough in practice since it just performs constraint concatenations.

Once the set $\{(\vec{i},\vec{j}) \mid \vec{i} \bowtie \vec{j}\}$ is obtained as a union of polytopes $= \cup \mathcal{Q}_k$, it remains to compute the set $DS = \{\vec{d} \mid \vec{d} = \vec{i} - \vec{j} \wedge \vec{i} \bowtie \vec{j}\}$. Adding the constraint $\vec{d} = \vec{i} - \vec{j}$ to $\mathcal{Q}_k$, we obtain an integral polyhedron $\mathcal{P}_k$ in a vector space of dimension $\dim \vec{i} + \dim \vec{j} + \dim \vec{d} = 3 \dim \vec{i}$. It remains to project each $\mathcal{P}_k$ on $\vec{d}$ to obtain DS, $DS = \cup_{k=1}^{r} \Pr(\mathcal{P}_k, \vec{d})$. In general, DS is not a polytope as required by the heuristics described in Section 4. We thus compute its convex hull, which is an over-approximation, thanks to Polylib [16].

Main example (cont'd) We automatically get the set DS of conflicting differences $\vec{d} = (d_1, d_2)$ for array y depicted in Figure 2.(b). It is defined by the following constraints:

$$\begin{cases} -(N-1) \leq d_1 + 2d_2 \leq N-1 \\ -(N-1) \leq d_1 - d_2 \leq N-1 \\ -1 \leq d_2 \leq 1 \\ -(N-1) \leq 2d_2 \leq N-1 \end{cases}$$

It is possible to find this set by analyzing the program by hand. However, even for such a simple example, the computation of DS can be very tricky. Thus, even in the case where the designer wants to choose the modular mapping him/herself, instead of relying on heuristics, it is very useful to be able to use an automated analysis to compute this fundamental (for lattice-based memory allocation) object DS. This is what the first part of our tool provides. ■

4 Deriving Strictly Admissible Lattices

This section describes the algorithms we use in the second part of our tool, a stand-alone program devoted to polyhedral optimizations, to find good admissible lattices for a 0-symmetric polytope. We bridge the gap between the abstract description of the heuristics given in [5, 6] and an actual implementation. Currently, we use our program not only to find good mappings automatically and evaluate previously-proposed heuristics, but also to invent new heuristics and help us finding optimal mappings in a *semi-automatic* way. For example, we mentioned that the optimal modular mapping for the array `y` in Section 2 has memory size $2N-3$. Even if heuristics are good enough here (with memory size $2N$), finding the optimal in a parametric way is harder: we used our tool to find the optimal for a few values of N, then proved by hand that $2N-3$ is optimal. The same is true for triangular domains, see the example hereafter. In other words, this tool should be viewed as a platform for the development, understanding, and exploration of mappings.

As explained in Section 2, finding a valid modular mapping $(M, \vec{b})$, for a set of conflicting indices $DS \subseteq \mathbb{Z}^n$, amounts to find a strictly admissible integral lattice Λ for DS, i.e., a sublattice of $\mathbb{Z}^n$ whose intersection with DS is reduced to $\{0\}$. By construction, DS is 0-symmetric and is described or over-approximated as the integer points within a polyhedron K. This polyhedron K is described either by its vertices or by a polyhedral representation $\{\vec{x} \mid A\vec{x} \leq \vec{c}\}$. Currently, our tool has the following functionalities:

- Computation of optimal strictly admissible integer lattice by exhaustive search with the approach suggested in [6].
- Computation of the successive minima and of a set of corresponding minimum vectors.
- Computation of the gauge functions F_i and F_i^*;
- Implementation of the generalized basis reduction [14];
- Implementation of the different heuristics of [5].

We now explain the key algorithmic points underlying these developments, focusing only the non-obvious ones.

4.1 Rogers' Heuristic

The first heuristic proposed in [5, 6] to get a strictly admissible integer lattice is an adaptation of a mechanism due to Rogers [11], for a 0-symmetric polytope K in dimension n.

Heuristic 1

- *Choose n positive integers ρ_i, s.t. ρ_i is a multiple of ρ_{i+1}, and $\dim(\mathcal{L}_i) \leq i-1$, where $\mathcal{L}_i = Vect(K/\rho_i \cap \mathbb{Z}^n)$.*
- *Choose a basis $(\vec{a}_i)_{1\leq i\leq n}$ of $\mathbb{Z}^n$ s.t. $\mathcal{L}_i \subseteq Vect(\vec{a}_1, \ldots, \vec{a}_{i-1})$.*
- *Define Λ the lattice generated by the vectors $(\rho_i \vec{a}_i)_{1\leq i\leq n}$.*

The correctness and worse-case quality of such a strategy are analyzed in [5, 6]. It gives a correct mapping when $\rho_i \lambda_i(K) > 1$ with $\lambda_i(K)$ the i-th successive minimum of K:

$$\lambda_i(K) = \min\{\lambda > 0 \mid \dim(\mathrm{Vect}(\lambda K \cap \mathbb{Z}^n)) \geq i\}$$

We now explain how we compute the scaling factors ρ_i and the basis $(\vec{a}_i)_{1 \leq i \leq n}$. The first difficulty is to compute the successive minima. These minima are also useful to get the "dimension of the problem", i.e., the dimension $p \leq n$ of the vector space generated by the integer points in K. Indeed, a complete memory folding can be done for the $n - p$ other dimensions (i.e., each modulus is 1), while the p "full" dimensions may need a folding with modulus greater than 1. This mechanism generalizes the projections used in [18].

4.1.1 Computing the Successive Minima

The first successive minimum $\lambda_1(K)$ is the smallest rational number λ such that the polyhedron K/λ contains an integer point $\vec{x} \neq 0$, i.e., it is defined by:

$$\lambda_1(K) = \min\{\lambda \in \mathbb{Q} \mid \exists \vec{x} \in \mathbb{Z}^n, \vec{x} \neq 0, A\vec{x} \leq \lambda \vec{c}\}$$

The constraint $\vec{x} \neq 0$ makes this system not linear. But, if there exists such an $\vec{x}$, then $x_i \geq 1$ or $x_i \leq -1$ for some i. Moreover, as K is 0-symmetric, if $\vec{x} \in K$, then $-\vec{x} \in K$, so we can look only for $\vec{x}$ such that $x_i \geq 1$ for some i. In other words, we can compute $\lambda_1(K)$ as the minimum over n values, each computed by mixed integer linear programming:

$$\lambda_1(K) = \min_{1 \leq i \leq n} \min\{\lambda \in \mathbb{Q} \mid \exists \vec{x} \in \mathbb{Z}^n, x_i \geq 1, A\vec{x} \leq \lambda \vec{c}\}$$

This gives $\lambda_1(K)$ as well as a corresponding solution $\vec{x}_1$.

Main example (cont'd) We explained how DS can be computed in an exact way, even when N is a parameter, as the integer points within the polytope K (when $N \geq 3$):

$$\begin{aligned} K = \{(i_1, i_2) \mid & -(N-1) \leq i_1 + 2i_2 \leq (N-1), \\ & -(N-1) \leq i_1 - i_2 \leq (N-1), -1 \leq i_2 \leq 1\} \end{aligned}$$

see Figure 2.(b). The first minimum of K is $1/(N-1)$, obtained when looking for a point $\vec{x}$ such that $x_1 \geq 1$. Notice that, here, we give the solution in a parametric way but, as we will explain, our current implementation for the computation of successive minima assumes N is fixed. ■

Now, suppose that $\lambda_1(K), \ldots, \lambda_{i-1}(K)$ have been computed, with their corresponding $\vec{x}_1, \ldots, \vec{x}_{i-1}$. The i-th minimum is the smallest λ such that K/λ contains an integer point $\vec{x}$, linearly independent with $\vec{x}_1, \ldots, \vec{x}_{i-1}$. To express this condition, we first compute a basis of $\mathbb{Z}^n$ whose first $(i-1)$ vectors span the same space as $\vec{x}_1, \ldots, \vec{x}_{i-1}$. For that, we define the $n \times (i-1)$ matrix X with columns $\vec{x}_1, \ldots, \vec{x}_{i-1}$, and we compute the Hermite form [15] of X: $X = QH$ where Q is a $n \times n$ unimodular matrix and H is a $n \times (i-1)$

matrix whose top $(i-1) \times (i-1)$ submatrix is nonnegative, upper triangular, and the rest of H is zero. We then make the change of basis $\vec{x} = Q\vec{y}$. In this representation, $\vec{x}$ is linearly independent with $\vec{x}_1, \ldots, \vec{x}_{i-1}$ if and only if $y_j \geq 1$ or $y_j \leq -1$ for some $j \geq i$. Again because K is 0-symmetric, we have:

$$\lambda_i(K) = \min_{i \leq j \leq n} \min\{\lambda \in \mathbb{Q} \mid \exists \vec{y} \in \mathbb{Z}^n, y_j \geq 1, AQ\vec{y} \leq \lambda\vec{c}\}$$

Solving this system, we get $\lambda_i(K)$, a corresponding solution $\vec{y}_i$ and, finally, $\vec{x}_i = Q^{-1}\vec{y}_i$. Continuing this way, we obtain all successive minima $\lambda_i(K)$, one after the other, by solving $n(n+1)/2$ mixed integer linear programs.

4.1.2 Mixed Integer Linear Programming

For computing the successive minima with mixed integer linear programming, we tried two implementations that work for a non-parameterized 0-symmetric polytope K. Both compute of course the same successive minima $\lambda_i(K)$ but possibly with different $\vec{x}_i$'s as these are not uniquely defined.

Our first implementation uses the public-domain tool GLPK [10] and is easy to interface. The only problem is that the rational numbers $\lambda_i(K)$ are returned as float numbers, and not as the quotient of two integers. However, as each $\vec{x}_i$ is an integer vector, we use a post-processing step to compute $\lambda_i(K)$ as a fraction: it is the smallest rational number such that $A\vec{x}_i \leq \lambda\vec{c}$. We thus consider each equation of this system to build $\lambda_i(K)$ as the quotient of two integers.

Our second implementation uses PIP [8, 16], a *parametric* linear programming solver, which gives the lexicographic minimum in a parameterized polyhedron $\{A\vec{x} \leq B\vec{n} + \vec{c}\}$ where $\vec{x}$ represents nonnegative unknowns and $\vec{n}$ represents nonnegative integer parameters. Using PIP is a bit tricky. Indeed, some pre-processing must be done to cope with the fact that unknowns are nonnegative, but this is just a technical problem. More important is that, here, we want to use PIP for mixed integer linear programming. For that, as explained in [8], the first step is to consider that integer unknowns ($\vec{x}$ in our system) are parameters and find, with parametric (rational) linear programming, the minimum of λ as a function of $\vec{x}$. The solution is a QUAST, where each leaf u of the tree gives a solution $\lambda^{(u)}$ as an affine expression $f^{(u)}(\vec{x})$ of $\vec{x}$, valid when $\vec{x}$ belongs to some polyhedron $\mathcal{P}^{(u)}$. The second step finds, for each leaf u, thanks to *integer* linear programming, the minimum of $f^{(u)}(\vec{x})$ over all integer elements $\vec{x}$ of $\mathcal{P}^{(u)}$. The minimum of these minima gives the desired minimum λ.

Main example (cont'd) Suppose $N = 100$ for example. Our GLPK implementation first finds $\lambda_1(K) = 0.010101\ldots$ with $\vec{x}_1 = (1, 0)$, and $\lambda_1(K)$ is recomputed as $1/99$. Then, it finds $\lambda_2(K) = 1$ with $\vec{x}_2 = (0, 1)$. Our PIP implementation finds $\lambda_1(K)$ directly as the fraction $1/99$ for $\vec{x}_1 = (1, 0)$. Then, as PIP looks for the lexicographic minimum, it finds $\vec{x}_2 = (-98, 1)$, $\lambda_2(K) = 1$. Thus, the $\vec{x}_i$ may not be the same with GLPK or PIP, while the λ_i are uniquely defined. ■

When none of the mixed integer linear programs solved to get $\lambda_i(K)$ has a solution, K is not full-dimensional and we directly get, with no additional effort, the minimal vector space that contains all integer points of K. As previously mentioned, this is important to find a good lattice/mapping and not fall into traps due to skewed and flat polytopes K. As for complexity, although integer linear programming is NP-complete, the two implementations are very fast, at least for the practical cases that arise in our memory allocation problems, all of small dimensions. Compared to the time needed to analyze the program and compute K itself, running times are not worth mentioning. This is maybe the most important conclusion of our implementation study: *the limiting factor, in terms of complexity, for lattice-based memory allocation is the required program analysis, not the search for good admissible lattices*. Thus, the apparently heavy machinery proposed in [6] to get modular appings is only complicated mathematically, but it is not costly in practice.

What about parameterized polytopes? As mentioned, using PIP is bit more complicated than using GLPK. However, it has some parametric capabilities that GLPK does not have and our hope was to be able to derive $\lambda_i(K)$ (or at least $\vec{x}_i$) even if K is linearly parameterized as $\{A\vec{x} \leq B\vec{n} + \vec{c}\}$. Unfortunately, this does not work as λ is then a multiplicative factor of the parameters $\vec{n}$ in the linear programs that need to be solved. Looking for $1/\lambda$ instead of λ leads to the same difficulty as $1/\lambda$ is now a multiplicative factor of $\vec{x}$. Also, even if we were able to compute $\lambda_1(K)$, our technique to make sure that $\vec{x}_2$ and $\vec{x}_1$ are linearly independent would fail if $\vec{x}_1$ is parameterized too, unless we can complete $\vec{x}_1$ into a basis in a parametric way. So, this seems difficult. But $\lambda_1(nK) = \lambda_1(K)/n$, with the same minimum vector, so there must be a way to express, at least in some cases to be specified, each $1/\lambda_i(K)$ as a QUAST of parameters. Unfortunately, so far, we did not find a way to compute the successive minima of a linearly parameterized polyhedron.

4.1.3 How to Choose the $(\rho_i)_{1\leq i\leq n}$ and $(\vec{a}_i)_{1\leq i\leq n}$

Once the successive minima are computed, one can easily choose the scaling factors ρ_i and an adequate basis $(\vec{a}_i)_{1\leq i\leq n}$ of $\mathbb{Z}^n$, as required by Heuristic 1.

For the vectors $(\vec{a}_i)_{1\leq i\leq n}$, a possible choice, suggested in [5], is to use the vectors $\vec{x}_i$ associated with the successive minima $\lambda_i(K)$. Indeed, if $\rho_i\lambda_i(K) > 1$, then the vector space $\mathcal{L}_i = \mathrm{Vect}(K/\rho_i \cap \mathbb{Z}^n)$ has at most $(i-1)$ dimensions. Furthermore, $\mathcal{L}_i \subseteq \mathrm{Vect}(\vec{x}_1, \ldots, \vec{x}_{i-1})$. However, the vectors $(\vec{x}_i)_{1\leq i\leq n}$, although they form a basis of $\mathbb{Q}^n$, may not form a basis of $\mathbb{Z}^n$. To get a basis of $\mathbb{Z}^n$, we apply the same Hermite normal form trick that we used previously. We compute $X = QH$ and we define the vectors $\vec{a}_i$ as the columns of Q.

For the scaling factors ρ_i, a possibility is to choose ρ_i to be the smallest power of 2 strictly larger than $1/\lambda_i(K)$. It turns out however that the divisibility condition for the ρ_i in Heuristic 1 is often useless in practice. Most of the time, especially in small dimensions, it is enough to choose $\rho_i = \lfloor 1/\lambda_i(K) \rfloor + 1$ to get a (smaller) strictly admissible integer lattice. If not, our current implementation increments the ρ_i, one at a time, in a round-robin fashion, and checks if this gives a solution. We call this heuristic Heuristic 1.a. At each step (in general one or two), we need to check whether the selected lattice $(\rho_i\vec{a}_i)_{1\leq i\leq n}$

is strictly admissible for $K = \{A\vec{x} \leq \vec{c}\}$. This amounts to check that there is no $\vec{x} \neq 0$ such that $\vec{x} = \sum_{i=1}^{n} x_i \rho_i \vec{a}_i$ and $A\vec{x} \leq \vec{c}, \vec{x} \neq 0\}$, where each x_i is an integer. This can be done by solving n linear programs, using the same technique we used to compute λ_1.

Finally, it remains to compute a mapping whose kernel is the selected lattice. If Q is the matrix whose columns are $(\vec{a}_i)_{1 \leq i \leq n}$ and $\vec{b}$ the vector whose components are $(\rho_i)_{1 \leq i \leq n}$, a suitable mapping is $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ where $M = Q^{-1}$.

Main example (cont'd) For this example, a good mapping is particularly simple to find by hand because the progress of uses along the array `y` follows the canonical dimensions of the array. Let us see what our implementation finds. Assume that $N = 100$. With GLPK, we get $\lambda_1(K) = 1/99$ and $\lambda_2(K) = 1$ for $\vec{x}_1 = (1, 0)$ and $\vec{x}_2 = (0, 1)$. With Heuristic 1, we get $\rho_1 = 128$ and $\rho_2 = 2$ with the mapping $\sigma(i_1, i_2) = (i_1 \bmod 128, i_2 \bmod 2)$, which can be simplified into $\sigma(i_1, i_2) = (i_1, i_2 \bmod 2)$ as $0 \leq i_1 < N \leq 128$. With Heuristic 1.a, we will select directly $\rho_1 = 99 + 1 = N$, even if the initial array size in the first dimension is larger. In practice, Heuristic 1.a is more realistic. Going to the next power of 2 as in Heuristic 1 can lead to a loss in memory. ■

We point out that, in practice, one should pay attention to the "complexity" of the derived lattice and of the resulting mapping. So far, the only optimizing criterion was the size of the memory, i.e., all mappings leading to the same memory size are considered equally good. But some can lead to simpler access functions (without even mentioning cache access). A possibility to get simple lattices is to minimize, when looking for each $\vec{x}_i$, its components in absolute value. One can also use, after all $\vec{x}_i$ are computed, a form of basis reduction to get a simpler basis. One example is the generalized basis reduction of Lovász and Scarf [14] that we implemented. On random polytopes, we saw some improvements but this was not convincing on polytopes arising from simple programs. For Heuristic 1.(a), to preserve the successive vector spaces $\mathcal{L}_i$, one can limit the modifications to "upper triangular" changes, i.e., simplifying $\vec{x}_i$ with linear combinations of the previous $\vec{x}_j$, $j < i$. But how to derive the "simplest" mapping for a given lattice is still an issue.

Main example (cont'd) To illustrate these possible complications, consider the basis found by our implementation with PIP: $(1, 0)$ and $(-98, 1)$. This basis leads to the "dirty" mapping $\sigma(i_1, i_2) = (i_1 + 98i_2 \bmod 100, i_2 \bmod 2)$ with Heuristic 1.a. It can be simplified into $\sigma(i_1, i_2) = (i_1 - 2i_2 \bmod 100, i_2 \bmod 2)$ but, still, this is more complicated than the mapping found with GLPK. To get $\vec{x}_2 = (0, 1)$ as for GLPK instead of $\vec{x}_2 = (-98, 1)$, we need to complete our implementation so that it simplifies the basis. Looking for $\vec{x}_2$ with minimal components in absolute value would lead to $\vec{x}_2 = (0, 1)$. One can also simplify $\vec{x}_2$ with $\vec{x}_1$: $\mathcal{L}_2$ is preserved and this also leads to $(0, 1) = (-98, 1) + 98(1, 0)$. ■

4.2 Heuristics based on gauge functions

We tried different heuristics on thousands of 0-symmetric polytopes that we randomly generated. Heuristic 1.a, the variant of Heuristic 1 explained in the previous section, is, on

average (but not always), the one that leads to the smallest lattices. However, as explained in the previous section, it requires the computation of the successive minima – with mixed integer linear programming – and the computation of adequate scaling factors – with integer linear programming – if we choose ρ_i in a more accurate way than just the smallest acceptable power of 2. Although these computations may appear very expensive, for the practical cases arising from programs, this can be done very quickly so using Heuristic 1.a is possible in practice. However, as mentioned, we do not know how to use this heuristic for parameterized polytopes and this may be a problem for practical memory reuse when optimizing programs. This is less true for high-level synthesis where the parameters are in general known at compile-time although using parametric techniques as late as possible in the transformation process would be a plus.

Unlike Heuristics 1 and 1.a, the two heuristics that we describe in this section can be easily parameterized. These are the heuristics proposed in [5, 6] as generalizations of the heuristic of Lefebvre and Feautrier [13]. They lead to memory allocations of the form $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ where $\vec{b}$ can be linearly parameterized if K itself is linearly parameterized. However, one needs to give them, as input, a non-parameterized basis, or alternatively, a non-parameterized matrix M. Before, let us recall some definitions from [5, 6].

The function $F(\vec{x}) = \min\{\lambda > 0 \mid \vec{x} \in \lambda K\}$ defines a norm such that $F(\alpha\vec{x}) = |\alpha|F(\vec{x})$, called the gauge function of K. Given some vectors $(\vec{a}_i)_{1\leq i\leq n}$, one can define $F_i(\vec{x}) = \min\{F(\vec{y}) \mid \vec{y} \in \vec{x} + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\}$, which is connected to the gauge function of the projection of K along the vectors $\vec{a}_1, \ldots, \vec{a}_{i-1}$. The next heuristic uses the functions F_i.

Heuristic 2

- *Choose n linearly independent integral vectors $(\vec{a}_1, \ldots, \vec{a}_n)$.*
- *Compute $F_i(\vec{a}_i) = \min\{F(\vec{y}) \mid \vec{y} \in \vec{a}_i + \mathit{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\}$, for $1 \leq i \leq n$.*
- *Choose n integers ρ_i such that $\rho_i F_i(\vec{a}_i) > 1$.*
- *Define Λ the lattice generated by the vectors $(\rho_i\vec{a}_i)_{1\leq i\leq n}$.*

Again, we refer to [5, 6] for the correctness and worse-case quality of such a strategy. We focus here on the implementation details. The only algorithmic need is to compute $F_i(\vec{a}_i)$ for all $1 \leq i \leq n$. Here is how we proceed. We have:

$$\begin{aligned} F_i(\vec{a}_i) &= \min\{F(y) \mid \vec{y} \in \vec{a}_i + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\} \\ &= \min\{\lambda > 0 \mid \vec{y} \in \lambda K,\ \vec{y} = \vec{a}_i + \sum_{j=1}^{i-1} y_j\vec{a}_j\} \\ &= \min\{\lambda > 0 \mid A\vec{y} \leq \lambda\vec{c},\ \vec{y} = \vec{a}_i + \sum_{j=1}^{i-1} y_j\vec{a}_j\} \end{aligned}$$

If K is not parameterized, the last expression can be solved with (rational) linear programming. Otherwise, if $K = \{A\vec{x} \leq B\vec{n} + \vec{c}\}$, then λ is a multiplicative factor of the parameters and PIP cannot be directly used. But:

$$F_i(\vec{a}_i) = \min\{\lambda > 0 \mid \vec{y} \in \lambda K,\, \vec{y} = \vec{a}_i + \sum_{j=1}^{i-1} y_j \vec{a}_j\}$$

and, with $\rho = 1/\lambda$ and $\vec{z} = \rho\vec{y}$, we have

$$\begin{aligned} F_i(\vec{a}_i) &= \min\{1/\rho > 0 \mid \vec{z} \in K,\, \vec{z} = \rho\vec{a}_i + \sum_{j=1}^{i-1} z_j \vec{a}_j\} \\ &= 1/\max\{\rho > 0 \mid A\vec{z} \leq B\vec{n} + \vec{c},\, \vec{z} = \rho\vec{a}_i + \sum_{j=1}^{i-1} z_j \vec{a}_j\} \end{aligned}$$

The last expression shows that, using PIP, we can find the inverse of $F_i(\vec{a}_i)$ with rational parametric linear programming, even for a linearly-parameterized polytope K.

Once all $F_i(\vec{a}_i)$ are computed, we let $\rho_i = \lfloor 1/F_i(\vec{a}_i) \rfloor + 1$. If $(\vec{a}_i)_{1 \leq i \leq n}$ defines a basis of $\mathbb{Z}^n$, again, the mapping is $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ where M is the inverse of the matrix whose columns are the $\vec{a}_i$ and $\vec{b}$ is the vector defined by $b_i = \rho_i$.

Main example (cont'd) For our running example, with $\vec{a}_1 = (1, 0)$ and $\vec{a}_2 = (0, 1)$, we find $1/F_1(\vec{a}_1) = N - 1$ and $\rho_1 = N$, then $F_2(\vec{a}_2) = 1$ and $\rho_2 = 2$, with the mapping $\sigma(i_1, i_2) = (i_1 \bmod N, i_2 \bmod 2)$. Also, as mentioned in [5], with a slight basis change, we can also obtain a valid $1D$ mapping as $\sigma(i_1, i_2) = 2i_1 + i_2 \bmod 2N$. For this particular case, we are not far from the optimal $\sigma(i_1, i_2) = 2i_1 + i_2 \bmod 2N - 3$. Picking the vectors in the opposite order leads to $\rho_1 = 2$, $\rho_2 = N$ and the mapping $(i_2 \bmod 2, i_1 \bmod N)$. We can also get the valid $1D$ mapping $Ni_2 + i_1 \bmod 2N$. ■

An alternative view of Heuristic 2 is Heuristic 3 below, which directly builds a valid mapping, instead of a strictly admissible integer lattice. This is Lefebvre and Feautrier's approach, generalized to an arbitrary set of independent vectors $(\vec{c}_i)_{1 \leq i \leq n}$. The details of the connection with Heuristic 2 is explained in [5]. This is a dual view of the same approach. The function F_i^* is the equivalent of F_i but for K^*, the polar reciprocal of K, which is $K^* = \{\vec{y} \mid \vec{x}.\vec{y} \leq 1 \text{ for all } \vec{x} \in K\}$. For the implementation, we use the alternative definition of $F_i^*(\vec{c}_i)$ as $F_i^*(\vec{c}_i) = \sup\{\vec{c}_i.\vec{x} \mid \vec{x} \in K, \vec{c}_j.\vec{x} = 0, \forall j < i\}$.

Heuristic 3

- *Choose n linearly independent integer vectors $(\vec{c}_1, \ldots, \vec{c}_n)$.*
- *Compute $F_i^*(\vec{c}_i) = \sup\{\vec{c}_i.\vec{x} \mid \vec{x} \in K, \vec{c}_j.\vec{x} = 0, \forall j < i\}$, $1 \leq i \leq n$.*
- *Choose n integers ρ_i such that $\rho_i > F_i^*(\vec{c}_i)$.*
- *Let M be the matrix with row vectors $(\vec{c}_i)_{1 \leq i \leq n}$ and $\vec{b}$ the vector such that $b_i = \rho_i$.*

Kernel	Array		Storage mapping found		Method				Runtime (s)	
	Identifier	Original	Mapping	Compressed	H1	H2	H3	OPT	DS	OPT
durbin.c	alpha	100	$i \mapsto i \bmod 1$	1	×	×	×	×	0.1	0.3
	beta	100	$i \mapsto i \bmod 1$	1	×	×	×	×	0.1	0.002
	sum	10000	$(i,j) \mapsto (i \bmod 1, j \bmod 1)$	1	×	×	×	×	0.5	0.003
	y	10000	$(i,j) \mapsto (i \bmod 100, j \bmod 2)$	200	×	×	×		1.8	
			$(i,j) \mapsto (i \bmod 1, 2i + j \bmod 197)$	197				×		17
reg_detect.c	sum_t	36	$(i,j) \mapsto (i \bmod 6, j \bmod 6)$	36	×	×	×		0.1	
			$(i,j) \mapsto (i \bmod 3, i + j \bmod 9)$	27				×		0.2
	mean	36	$(i,j) \mapsto (i \bmod 6, j \bmod 6)$	36	×	×	×		0.07	
			$(i,j) \mapsto (i \bmod 3, i + j \bmod 9)$	27				×		0.3
	diff	2304	$(i,j,k) \mapsto (i \bmod 6, j \bmod 6, k \bmod 64)$	2304	×	×	×		0.1	
			$(i,j,k) \mapsto (i \bmod 3, i + j \bmod 9, k \bmod 64)$	1728				×		–
	sum_d	2304	$(i,j,k) \mapsto (i \bmod 1, j \bmod 1, k \bmod 1)$	1	×	×	×	×	0.7	0.004
dynprog.c	c	100	$(i,j) \mapsto (i \bmod 9, j \bmod 9)$	81	×	×	×		0.5	
			$(i,j) \mapsto (i \bmod 1, 13i + j \bmod 61)$	61				×		0.6
	sum_c	1000	$(i,j,k) \mapsto (i \bmod 1, j \bmod 1, k \bmod 1)$	1	×	×	×	×	6.1	0.004
gauss.c	g_acc1	10000	$(i,j,k) \mapsto (i \bmod 1, j \bmod 1, k \bmod 1)$	1	×	×	×	×	0.7	0.01
	g_acc2	10000	$(i,j,k) \mapsto (i \bmod 1, j \bmod 1, k \bmod 1)$	1	×	×	×	×	0.9	0.007
	g_tmp	2500	$(i,j) \mapsto (j \bmod 50, i \bmod 48)$	2400	×			×	0.07	46 min
			$(i,j) \mapsto (i \bmod 48, j \bmod 50)$	2400		×	×	×		|
			$(i,j) \mapsto (j - i \bmod 2, 24j - 25i \bmod 1200)$	2400				×		|
mot_detect_kern.c	Delta	68121	$(i,j,k) \mapsto (k \bmod 10, j \bmod 1, i \bmod 1)$	10	×			×	3.9	0.3
			$(i,j,k) \mapsto (i \bmod 1, j \bmod 1, k \bmod 10)$	10		×	×	×		|
	ODelta	842	$i \mapsto i \bmod 1$	1	×	×	×	×	0.4	0.002

Figure 3: Experimental results for the kernels given in appendix. The time spent in the lifetime analysis (DS) and in the optimal method (OPT) are given for a Pentium III CPU 800Mhz with 256 MB RAM.

This time, it is easy to see that, with no additional transformation, $F_i^*(\vec{c}_i)$ can be computed with parametric (rational) linear programming if K is linearly parameterized. In other words, as the function F_i and F_i^* are the inverse of each other (see [5] for more details), we can implement Heuristics 2 and 3 if either K or K^* is linearly parameterized, for a non-parameterized basis. We implemented both heuristics (which give, by nature, the same results for dual basis $(\vec{a}_i)_{1 \le i \le n}$ and $(\vec{c}_i)_{1 \le i \le n}$, when vectors are picked in the opposite order), trying different basis as input: identity, identity up to a permutation of rows or columns (this can be viewed as a generalization of the technique of [7]), basis given by the successive minima, basis given by generalized basis reduction, etc. All these techniques are fast enough to allow such attempts. We then pick the best solution.

5 Experiments

Our stand-alone mathematical tool Cl@k (Section 4) provides the heuristics of [5, 6] (and some other) while our program analyzer tool Bee (Section 3) makes the link with programs, i.e., computes the lifetime analysis of array elements required by the heuristics and generates the final C program with the allocations. Several libraries are involved, including ROSE [19], Polylib and PIP [16]. ROSE is a compiler framework providing simple mechanisms to read and write a program abstract syntax tree. It uses the SAGE intermediate representation and exploits ideas of the Nestor library [21]. We use ROSE to extract iteration domains $\mathcal{I}_S$ and array index functions from static control programs written in C. Both AST traversals (top-down and bottom-up) and rewrite facilities are consequently used in our tool. The polyhedral operations required by the different steps of our method are computed thanks to the Polylib library [16]. Finally, the lexicographic minima/maxima of integral polytopes are derived thanks to the PIP library [8, 16].

We applied our tools to contract temporary arrays of several image processing kernels [31] [1]. We chose these kernels as they are used in all related papers and because they are short enough to be given here (see appendix). Figure 3 gives the results of our experiments. The benchmarks gather (1) our main example with $N = 100$ (durbin.c), (2) a real-time regularity detection used in robot vision, (3) a toy example similar to dynamic programming, (4) a 2D Gaussian blur filter used in image processing, (5) a motion detection algorithm used in the transmission of real time videos on data networks. The "Array" column gives the temporary arrays to contract. The "Mapping" column shows the final allocation σ found by the different methods. The mapping definitions assume that the operator mod has the weakest priority, i.e., $i + j$ mod 10 stands for $(i + j)$ mod 10. The "Compressed" column gives the array size after remapping, i.e., the product of the moduli. The "Runtime" column gives, for each array, the runtime for the computation of lifetime analysis (DS) and the runtime for the exhaustive search leading to size-optimal mappings (OPT). We now analyze these results and draw some conclusions from this study.

In terms of complexity, we point out again that the main factor of the complete strategy is the analysis itself. The runtimes of the different heuristics are so small they are not worth mentioning. The search for an optimal mapping is, however, too costly for large sizes. Indeed, this search is not optimized, it simply checks all determinants starting from a pre-computed lower bound and the number of determinants to consider can become very large. We also point out that, although previous approaches do not use the concept of DS (the search for the mapping is often combined with the analysis itself), they all use a lifetime analysis that is not cheaper than the computation of DS.

In terms of contraction, it appears that most temporary arrays can be contracted, leading to a very good compression ratio. Our study also confirms that, for some practical cases, the various heuristics give a similar (if not the same) result and that one can even restrict to a simple basis such as the identity matrix (or a permutation of it). This is because the loop indices (the "schedule") are often aligned with the array indices. We also point out that the principles used in Heuristics 2 and 3 are combinations of the principles of the heuristics of [13] (to get the moduli) and of [18] (to get the right projection when DS is not full-dimensional) and that, again on simple examples, results can thus be similar. But there is a fundamental difference: we can apply the heuristics based on DS even if the initial program is not in single-assignment form and even if there are more than one statement writing a given array. For example, in [18], there must be, by principle, a one-to-one correspondence between arrays and statements. Indeed, the mapping is derived by a matrix relation between the unique schedule and the unique access function corresponding to an array. If the array is written by several statements, even if each array element is written once, the program needs to be changed so that each statement writes in a separate memory. For example, in Durbin's kernel, S_4 and S_3 would need to write in two different arrays, unless S_4 is pushed into the previous loop (inverse of loop peeling) and converted into an instance of S_3. This is possible here, but certainly not in general. Therefore, a technique based on the set DS seems superior because it can handle more programs

[1] Many thanks to Florin Balasa who gave us these code examples.

(even parameterized ones) and because it gives more freedom to compute the mapping as the program analysis is decoupled from the computation of the strictly admissible integer lattice.

A particular form of array contraction is array scalarization, when an array can be completely transformed into a scalar. This is a particular case of our memory reduction technique. Indeed, an array can be scalarized if and only if its corresponding DS is reduced to $\{0\}$. Also a modulus equal to 1 corresponds to a dimension removal. We let the reader check that, for the codes given in appendix many arrays have been scalarized. The only case we miss is for the kernel (5). This is due to the fact that DS is over-approximated due to integer divisions in the derived clauses, which prevents us to see that DS is indeed equal to $\{0\}$. We know how to handle such cases correctly though it is not available yet in our current implementation.

Finally, the results in Figure 3 indicate that, although the heuristics are quite good in order of magnitude, there is still some space for improvement compared to the optimal, especially when DS has a strange shape. We have already mentioned the case of the array `y` in our running example, but the difference between N, the size found by the heuristics, and the optimal size $2N-3$ is not worth the price of the complicated access function. However, consider the kernel (2) and the different triangular-shaped arrays `mean`, `sum_t`, and `diff`. Although these arrays could be removed if some loop fusion was done, let us focus here on reducing their size without changing the program schedule.

Consider the 2D array `mean` for example. Its size is N^2 but only $N(N+1)/2 \sim N^2/2$ elements are live. They all conflict since they are all written before any is read. This defines the set $DS = \{(i,j) \mid |i| < N, |j| < N, |i-j| < N\}$. Surprisingly, none of the heuristics proposed here and in previous papers is able to find any reduction. But some reduction is possible with a modular mapping, leading to a memory size, not of order $N^2/2$, but of order $3N^2/4$. An optimal mapping is $(i,j) \mapsto (i+j \bmod 3m, j \bmod m)$ if $N = 2m$ and $(i,j) \mapsto (i-(3m+2)j) \bmod (3m^2+3m+1)$ if $N = 2m+1$. How can we find such a mapping? First, we can use our exhaustive search to find optimal mappings for small values of N. If some pattern appears, we can try to generalize the mappings, then prove their optimality. This how we proceeded here, thus in a *semi-automatic* process.

We can go further and use the same principle to *automatically* derive good parametric mappings. Indeed, pick a small value of N, for example 1. Find an optimal strictly admissible lattice for $DS = \{A\vec{x} \leq \vec{b}\}$. For the array `mean`, we get the lattice Λ generated by $(3,0)$ and $(1,-1)$, with determinant 3. Multiplying by λ, we get that $\lambda\Lambda$ is strictly admissible for $\{A\vec{x} \leq \lambda\vec{b}\}$. For the array `mean` and $\lambda = N-1$, we get a memory size $3(N-1)^2$, which is not what we want. But we can be more subtle: if Λ is strictly admissible for $\{A\vec{x} \leq \vec{b}\}$, it is weakly admissible (i.e., boundary intersection is acceptable) for $\{A\vec{x} \leq \vec{b}+\vec{1}\}$, which is here $\{A\vec{x} \leq 2\vec{b}\}$. Multiplying by m, we get that $m\Lambda$ is weakly admissible for $\{A\vec{x} \leq 2m\vec{b}\}$ and thus strictly admissible for $\{A\vec{x} < N\vec{b}\}$. We retrieve the optimal mapping for $N = 2m$ with size $3m^2$. The same mapping is valid for $N = 2m-1$ with size $\sim 3N^2/4$, though not optimal. In future work, we plan to use such a generalization principle to derive good parametric mappings.

6 Conclusion

We have presented a complete and effective compile-time analysis to contract arrays. An element-wise lifetime analysis for arrays has been proposed that computes the first (resp. last) statement instance writing (resp. reading) a generic array cell. We have shown how this algorithm can be used to build the set of conflicting array cells, required by the contraction methods described in [6], for which we proposed a complete and efficient implementation. Experimental results on a few kernels are provided and show important compression ratios, confirming the efficiency of the method. The whole analysis takes a few seconds for each benchmark on a Pentium III CPU 800MHz, with 256MB RAM, and most of the execution time is spent in lifetime analysis. We point out that computing last reads is more expensive that computing first writes, as there are more reads than writes in general. Some work needs to be done to accelerate this process.

Our method is however restricted to programs with affine array index functions and loops. In future work, we plan to address general programs by providing a conservative lifetime analysis (containing the real lifetime). Our lifetime analysis is actually able to provide lifetimes depending on a parameter (typically, N in the running example). Unfortunately, part of our current implementation is not yet able to handle them. We also propose to address this point in a future work. Finally, as mappings obtained from different methods of [6] can impact on spatial data locality, it would be good to have a method to select the best one.

References

[1] F. Balasa, P. G. Kjeldsberg, M. Palkovic, A. Vandecappelle, and F. Catthoor. Loop transformation methodologies for array-oriented memory management. In *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 205–212, Washington, DC, USA, 2006.

[2] P. Clauss, F. J. Fernandez, D. Gabervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. Technical Report ICPS number 06-04, Université Louis Pasteur, Oct. 2006.

[3] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PLDI'95)*, Santa Barbara, CA, July 1995.

[4] A. Darte and G. Huard. New complexity results on array contraction and related problems. *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 40(1):35–55, 2005.

[5] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. In 6*th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*, pages 298–308, San Jose, USA, Oct. 2003.

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

[7] E. De Greef, F. Catthoor, and H. De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[8] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

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

[10] GNU Free Software. GLPK (GNU Linear Programming Kit). http://www.gnu.org/software/glpk.

[11] P. M. Gruber and C. G. Lekkerkerker. *Geometry of Numbers*. North Holland, second edition, 1987.

[12] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, Sept. 2002.

[13] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[14] L. Lovász and H. E. Scarf. The generalized basis reduction algorithm. *Mathematics of Operations Research*, 17(3):751–764, 1992.

[15] M. Newman. *Integral Matrices*. Academic Press, 1972.

[16] PIP/Polylib: The Parametric Integer Programming's home. http://www.piplib.org.

[17] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, Aug. 1992.

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

[19] D. J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[20] J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. Reducing memory requirements of nested loops for embedded systems. In *DAC'01: Proceedings of the 38th conference on Design automation*, pages 359–364, 2001.

[21] G.-A. Silber and A. Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, Apr. 1999.

[22] Y. Song, R. Xu, C. Wang, and Z. Li. Improving data locality by array contraction. *IEEE Transactions on Computers*, 53(9):1073–1084, 2004.

[23] M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independent storage mapping for loops. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 24–33, San Jose, USA, 1998. ACM Press.

[24] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *International Conference on Programming Language Design and Implementation (PLDI'01)*, pages 232–242. ACM Press, 2001.

[25] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor. Storage size reduction by in-place mapping of arrays. In A. Cortesi, editor, *Verification, Model Checking and Abstract Interpretation, Third International Workshop, VMCAI 2002*, volume 2294 of *LNCS*, pages 167–181. Springer Verlag, 2002.

[26] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *7th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'04)*, pages 220–229, Washington DC, USA, Sept. 2004.

[27] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems*, 5(2):472–511, 2006.

[28] S. Verdoolaege, H. Nikolov, and T. Stefanov. Improved derivation of process networks. In *4th Workshop on Optimization for DSP and Embedded Systems (ODES-4)*, New York, USA, Mar. 2006.

[29] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, 1991.

[30] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC'99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 811–816, 1999.

[31] H. Zhu, I. I. Luican, and F. Balasa. Memory size computation for multimedia processing applications. In *ASP-DAC'06: Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 802–807, 2006.

A Source code of kernels

gauss.c

```
#define M 50
#define N 50
#define T 1
void main() {
  int x, y, k, tot[4];
  int in_image[N][M]; //live-in
  int Gauss[4]; // live-in
  int gauss_image[N][M]; // live-out
  int g_tmp[N][M];
  int g_acc1[N][M][4];
  int g_acc2[N][M][4];

  tot[0]=0;
  for (k=T-1; k<=1+T; k++)
    tot[k+2 - T] = tot[k+1 - T]
                 + Gauss[k - T];
  for (k=T-1; k<=1+T; k++)
    tot[k+2 - T] = tot[k+1 - T]
                 + Gauss[k - T];
  for(x=1; x<N-1; x++)
    for(y=0; y<M; y++) {
      g_acc1[x][y][0]=0;
      for(k=T-1; k<=1+T; k++) {
        g_acc1[x][y][k+2-T] =
          g_acc1[x][y][k+1-T]
          + in_image[x+k][y] * Gauss[k-T];
      }
      g_tmp[x][y] =
               g_acc1[x][y][3]/tot[3];
    }
  for(x=1; x<N-1; x++)
    for(y=1; y<M-1; y++) {
      g_acc2[x][y][0]=0;
      for(k=T-1; k<=1+T; k++) {
        g_acc2[x][y][k+2-T] =
          g_acc2[x][y][k+1-T]
          + g_tmp[x][y+k-T] * Gauss[k-T];
      }
      gauss_image[x][y] =
                  g_acc2[x][y][3]/tot[3];
    }
}
```

reg_detect.c

```
#define N 6
#define M 5
#define P 64
void main() {
  int i, j, k;
  int sum_t[N][N], mean[N][N];
  int diff[N][N][P], sum_d[N][N][P];
  int tangent[M]; // live-in
  int path[N][N]; // live-out

  for(j=0; j<=N-1; j++) {
    sum_t[j][j] = tangent[(N+1)*j];
    for( i=j+1; i<=N-1; i++)
      sum_t[j][i] = sum_t[j][i-1]
                  + tangent[i+N*j];
  }
  for(j=0; j<=N-1; j++)
    for(i=j; i<=N-1; i++)
      for(k=0; k<=P-1; k++)
        diff[j][i][k] = sum_t[j][i];
  for(j=0; j<=N-1; j++)
    for(i=j; i<=N-1; i++)
      sum_d[j][i][0] = diff[j][i][0];
      for(k=1; k<=P-1; k++)
        sum_d[j][i][k] =
                  sum_d[j][i][k-1]
                  + diff[j][i][k];
      mean[j][i] = sum_d[j][i][P-1];
  for(j=0; j<=N-1; j++)
    for(i=j; i<=N-1; i++)
      if (j>0) path[j][i] =
                  path[j-1][i-1]
                  + mean[j][i];
      else path[j][i] = mean[j][i];
}
```

mot_detect_kern.c

```
#define m 4
#define n 4
#define M 32
#define N 32
void main() {
  int i, j, k, l, ODelta[(M-m+1)*(N-n+1)+1];
  int Delta[M-m+1][N-n+1][(2*m+1)*(2*n+1)];
  int A[(m+1)*(m+1)][(n+1)*(n+1)]; // live-in
  int opt[1]; // live-out

  ODelta[0]=0;
  for(i=m; i<=M; i++)
    for(j=n; j<=N; j++) {
      Delta[i][j][0]=0;
      for(k=i-m; k<=i+m; k++)
        for(l=j-n; l<=j+n; l++)
          Delta[i][j][(2*n+1)*k-(2*n+1)*i
                      +l-j+(2*m*n+m+n+1)] =
            A[i][j] - A[k][l] +
            Delta[i][j][(2*n+1)*k-(2*n+1)*i
                        +l-j+(2*m*n+m+n)];
      ODelta[(N-n+1)*i+j-(m*N-m*n+m+n+1)] =
        Delta[i][j][(2*m+1)*(2*n+1)] +
        ODelta[(N-n+1)*i+j-(m*N-m*n+m+n)];
    }
  opt[0] = ODelta[(M-m+1)*(N-n+1)];
}
```

dynprog.c

```
#define N 10
void main() {
  int i, j, k;
  int W[N][N], c[N][N]; // live-in
  int sum_c[N][N][N];
  int out[1]; // live-out

  for(i=0; i<=N-2; i++)
    for( j=i+1; j<=N-1; j++) {
      sum_c[i][j][i] = 0;
      for(k=i+1; k<=j-1; k++)
        sum_c[i][j][k] = sum_c[i][j][k-1]
                       + c[i][k] + c[k][j];
      c[i][j] = sum_c[i][j][j-1] + W[i][j];
    }
  out[0] = c[0][N-1];
}
```