Bee+Cl@k: An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE
Christophe Alias, Fabrice Baray, Alain Darte

To cite this version:
Christophe Alias, Fabrice Baray, Alain Darte. Bee+Cl@k: An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’07), Jun 2007, San Diego, United States. 10.1145/1273444.1254778. hal-03106126

HAL Id: hal-03106126
https://hal.science/hal-03106126
Submitted on 11 Jan 2021

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.
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, Cl@k, 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. They can be used either for embedded program optimizations, e.g., to limit memory expansion introduced for parallelization, or in high-level synthesis, e.g., to design memories between communicating hardware accelerators.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers, Optimization

General Terms Algorithms, Experimentation, Theory

Keywords Memory reduction, source-to-source transformations, program analysis, lattices

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 [13, 27] 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 [28]), 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 [20], developed at the Lawrence Livermore National Labs. Intra-array memory reuse was first proposed at Leuven/IMEC [1, 8], 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 [14, 19, 24, 25], including some more work at Leuven/IMEC [26], until a more general framework was proposed in [6, 7] 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 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 mem-
ory 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 [31, 32, 2, 29, 3]. 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 [23, 5]) 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 [30], are also relevant here, even if they do not exactly target memory reduction. We cannot cite them all. An interesting recent work is [21], 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 multidimensional arrays, as introduced in [10] as static control programs. Each particular execution of a statement $S$ – an operation – is represented by $(S, \vec{i})$ where $\vec{i}$, the iteration vector, 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 $Z_S$, 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 [18, 4].

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 $\sigma(\vec{i}) = M\vec{i} \mod \vec{b}$, where $M$ is a matrix, $\vec{b}$ is a vector, and the modulo operation is applied component-wise. Such a mapping $\sigma$ 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 [6, 7]. This formalism generalizes the principles of the previously-proposed techniques, in particular, by De

![Figure 1. Overview of the method](image-url)

Greef et al. [8], Lefebvre and Feautrier [14], Quilleré and Rajopadhye [19], Strout et al. [24], Thies et al. [25]. 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 $\sigma$: we call this “interval” the lifetime of the array index. As pointed out in [7], when a given array location is written several times, we could be more accurate, distinguish the 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 an array indices is analyzed, we can define conflicting indices: two array indices $\vec{i}$ and $\vec{j}$ conflict (denoted by $\vec{i} \preceq \vec{j}$) if their lifetimes intersect. We will make sure that such indices are not mapped to the same memory location through $\sigma$, i.e., $\vec{i} \preceq \vec{j}, \vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$. If $DS$ denotes the set of all $\vec{d}$ such that $\vec{d} = \vec{i} - \vec{j}$, with $\vec{i} \preceq \vec{j}$, 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 $\sigma$). 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 $Z^n$ to $Z^p$ is a sublattice of $Z^n$. A lattice $\Lambda$ 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 $\Lambda$ and such that the product of the components of $\vec{b}$ (the memory size) is equal to the determinant of $\Lambda$ (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 \( L \), a matrix \( M \)
and a vector \( b \) so that the kernel of \( (M, b) \) is \( L \). This may
need some standard matrix computations (Hermite or Smith
normal forms). Heuristics developed in Step 3 can also give
directly the modular mapping. See [7] for more details, we
just use what is explained there. Finally, Step 5 consists in
replacing each access \( A(f(i)) \) to an optimized array \( A \) by
the reference \( A(r(f(i))) \). This, again, is done thanks to the
source-to-source transformer ROSE [20]. Some refinements
on \( \sigma \) can be done, which are mentioned in Sections 3 and 4.

Main example We will use, as running example, the clas-
csical Durbin’s kernel, which solves a Toeplitz system with \( N \)
unknowns. See its code in Figure 2.(a). The array \( x \) is an
input array, the array \( y \) is an output array, the arrays \( \text{alpha}, \beta, \text{sum}, \) and \( y \) are temporary arrays, thus subject to mem-
ory reduction. In this example, any sophisticated tool should
detect that arrays \( \text{alpha}, \beta, \text{sum}, \) and \( y \) can be replaced by
scalars. For array \( \text{sum} \), this check requires inter-loop analy-
sis. 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 \( \sigma(i_1, i_2) = (i_1, i_2 \mod 2) \).
In other words, it is correct to replace, in the code, all occurre-
ces \( y[c_1][c_2] \) by \( y[c_1][c_2 \mod 2] \). We will illustrate how to use
DS to find automatically such a mapping.

We point out that the best modular mapping for \( y \) is
\( \sigma(i_1, i_2) = 2i_1 + i_2 \mod (2N - 3) \), 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 [7]
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 our tool Bee
to compute the set DS of conflicting index differences,
needed to compute modular allocations. This amounts to
compute the conflict relation \( \approx \), 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 \( c \) is the “time interval” between
its initialization and its last read. Classical methods handle
arrays as scalar variables and make a too rough approxima-
tion of control flow to be used here. This section explains how
to compute the first operation writing \( c \) (its initialization)
and the last operation reading \( c \), for static control programs.
Our method is similar on many aspects to exact instance-
wise dataflow analysis and revisit some ideas of [10].

3.1.1 First Write of an Array Element
Consider a statement \( S \) in a static control program writing
\( A[u(i)] \) where \( i \) is the iteration vector of \( S \) and \( u \) an affine
function. Consider a cell \( c \) of \( A \) and \( W_S(c) \) the set of opera-
tions \((S, i)\) writing \( A[c] \). Each operation must be valid and
write \( A[c] \), i.e., \( W_S(c) = \{(S, i) : i \in IS, u(i) = c\} \). Writing
\( S_1, \ldots, S_n \) the statements assigning \( A \) and denoting by \( S \)
the execution order between operations, the first operation
writing \( A[c] \) is \( FW(c) = \min_S \bigcup_{S \subseteq S_1} W_S(c) \), rewritten as:

\[
FW(c) = \min(S) \{\min_W W_{S_1}(c), \ldots, \min_W W_{S_n}(c)\}
\]

Following [10], the sequencing predicate between two opera-
tions \((S, i)\) and \((T, j)\) can be defined thanks to the strict lex-
ographic order \( <_j \) between iteration vectors: \((S, i) < (T, j)\)
iff a) \( i[1..M] <_j \) \( i[1..M] \) or b) \( i[1..M] = j[1..M] \) 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,
\( \min_S W_S(c) \) boils down to compute the lexicographic mini-
num of the polyhedron \( \{i : i \in IS, u(i) = c\} \). When the
polyhedron depends on parameters (as \( c \) here), classical al-
gorithms of integer linear programming cannot be applied
and we need to use parametric integer programming tech-
niques [9]. The tool PIP [17] implements such techniques and
outputs a set of clauses, where each clause is a pair (polyhe-
dral domain, affine solution) that defines the lexicographic
minimum for a subset of the parameter space.

Main example (cont’d) By definition, the set \( W_{S_3}(c) \) of
instances of \( S_3 \) writing \( y[c_1][c_2] \), with \( c = (c_1, c_2) \), is:

\[
\{(S_3, k, i) : 1 \leq k \leq N - 1, 0 \leq i < k - 1, i = c_1, k = c_2\}
\]

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 solution:

\[
\min_S W_{S_3}(c_1, c_2) = \begin{cases} 
N \geq 2 \\
1 \leq c_2 \leq N - 1 \quad (S_3, c_2, c_1) \\
0 \leq c_1 \leq c_2 - 1
\end{cases}
\]

To make it simpler, here in and in the rest of the paper, we
only give the clauses providing a solution.

Since the different \( \min_S W_{S_i}(c) \) are computed, it remains
to combine them in order to find the global first \( FW(c) \)
as precise in Equation 1. This is achieved thanks to the
combination rules given in [10].

Main example (cont’d) Applying our analysis on the
array \( y \), we get the following clauses:

\[
FW(c_1, c_2) = \begin{cases} 
c_1 = 0, c_2 = 0 \quad (S_1, ) \\
0 \leq c_2 \leq N - 1 \quad (S_3, c_2, c_1) \\
0 \leq c_1 \leq N - 2 \quad (S_1, c_2, c_1) \\
c_1 < c_2 \\
0 \leq c_1 \leq N - 1 \quad (S_1, c_1) \\
0 \leq c_2 \leq N - 1 \quad (S_1, c_1) \\
c_1 = c_2
\end{cases}
\]

Here, the computation of the local first writes \( \min_S W_{S_i}(c) \)
is easy as \( y \) is in single assignment. In general, the main
difficulty is the combination of clauses to get the global first
write. This is also where the complexity of the analysis comes
from. One can also try to remove redundant constraints.

3.1.2 Last Read of an Array Element
Similarly to the first write, the last operation reading an
array cell is derived from local reading sets. For each assign-
Let us explain this result intuitively. First, note that, for each 
array cell 3.2 Computing the Conflict Relation
As explained in Section 2, two array indices  and  conflict if their lifetimes  and  intersect. In other words, the conflict relation  is:

\[ i \cong j \iff \text{FW}(i) \cap \text{LR}(j) \neq \emptyset \]

With the clauses

\[ \text{FW}(i) = \{ i \in D_k : \omega_k(i) \}_{h=1,p} \]

and

\[ \text{LR}(i) = \{ i \in D'_k : \omega'_k(i) \}_{k=1,q} \]

which define partitions of the index set, we get the relation  through a union of sets.

We consider all  sets  (many are empty) defined for  in \( \{1,\ldots,p\} \),  in \( \{1,\ldots,q\} \),  in \( \{1,\ldots,q\} \), by:

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

Then

\[ i \cong j \iff (i,j) \in \bigcup_{k=1}^{p} \bigcup_{l=1}^{q} \bigcup_{m=1}^{p} \bigcup_{n=1}^{q} C_{k,l,m,n} \]

Decomposing the predicate  into affine inequalities, we can write  itself as a union of sets, and more precisely of polytopes. One can show that  and  are always finite non-parametric values, of the same order than the number  of statements of the program. This consequently provides an \( O(n^p) \) 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 \( \{ (i,j) \mid i \cong j \} \) is obtained as a union of polytopes  and  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-memory allocation) object  . This is what the first part of our tool provides.

4. Deriving Strictly Admissible Lattices
This section describes the algorithms we use in our tool Cl@k (Critical Lattice Allocation Kernel), a stand-alone program devoted to the search for good admissible lattices for a symmetric polytope. We bridge the gap between the abstract description of the heuristics given in [6, 7] 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  in Section 2 has memory reading a given array cell, its last instance is computed, then the results are combined to get the global last read.

**Main example (cont’d)** Our algorithm finds automatically the following last read of \( y[c_1, c_2] \):

\[ \begin{aligned}
0 \leq c_1 \leq N - 2 \\
0 \leq c_2 \leq N - 2 \\
2c_1 > c_2 \\
0 \leq c_1 \leq N - 1
\end{aligned} \]

\[ \text{LR}(c_1, c_2) = \begin{cases}
0 \leq c_1 \leq N - 2 \\
0 \leq c_2 \leq N - 2 \\
2c_1 > c_2 \\
0 \leq c_1 \leq N - 1
\end{cases} \]

Let us explain this result intuitively. First, note that, for each \((k, i) \in \mathcal{S}_2 \), the read of \( y[i][k-1] \) in \( \mathcal{S}_2 \) is then repeated in \( \mathcal{S}_5 \). Thus, none of the instances of \( \mathcal{S}_2 \) can be a last read for \( y \). Now, let us focus on \( \mathcal{S}_3 \). For each \((k, i) \in \mathcal{S}_3 \), \( k < N - 1 \), the array cell \( y[i][k] \) written by \((i, k) \) is read twice, by \((i, k + 1) \) (for the reference \( y[i][k+1] \)) and \((i, k + 1, k - i) \) (for the reference \( y[i][k+1] \)). The last read of \( y[i][k] \) is thus \( \max \{ (i, k + 1, k - i), (i, k + 1, k - i) \} \), 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 \( 2i < k \) (second clause). Furthermore, with \((k, i) \in \mathcal{S}_3 \), \( k = N - 1 \), the array cell \( y[i][k] \) will be read in \( \mathcal{S}_3 \) (last clause).

3.2 Computing the Conflict Relation

As explained in Section 2, two array indices  and  conflict if their lifetimes  and  intersect. In other words, the conflict relation  is:

\[ i \cong j \iff \text{FW}(i) \cap \text{LR}(j) \neq \emptyset \]

With the clauses

\[ \text{FW}(i) = \{ i \in D_k : \omega_k(i) \}_{h=1,p} \]

and

\[ \text{LR}(i) = \{ i \in D'_k : \omega'_k(i) \}_{k=1,q} \]

which define partitions of the index set, we get the relation  through a union of sets. We consider all \( p^2 \) sets  (many are empty) defined for \( k \in \{1,\ldots,p\} \),  in \( \{1,\ldots,q\} \),  in \( \{1,\ldots,q\} \), by:

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

Then

\[ i \cong j \iff (i,j) \in \bigcup_{k=1}^{p} \bigcup_{l=1}^{q} \bigcup_{m=1}^{p} \bigcup_{n=1}^{q} C_{k,l,m,n} \]

Decomposing the predicate  into affine inequalities, we can write  itself as a union of sets, and more precisely of polytopes. One can show that  and  are always finite non-parametric values, of the same order than the number  of statements of the program. This consequently provides an \( O(n^p) \) 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 \( \{ (i,j) \mid i \cong j \} \) is obtained as a union of polytopes  and  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-memory allocation) object  . This is what the first part of our tool provides.

4. Deriving Strictly Admissible Lattices
This section describes the algorithms we use in our tool Cl@k (Critical Lattice Allocation Kernel), a stand-alone program devoted to the search for good admissible lattices for a symmetric polytope. We bridge the gap between the abstract description of the heuristics given in [6, 7] 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  in Section 2 has memory

Figure 2. (a) Durbin’s kernel, (b) Set DS of conflicting index differences for array y (polytope), an optimal integer lattice A (grey points), and a corresponding mapping σ (all other temporary arrays are scalableizable), (c) Transformed program.
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 $(\bar{M}, \bar{b})$, for a set of conflicting indices $DS \subseteq \mathbb{Z}^n$, amounts to find a strictly admissible integral lattice $\Lambda$ 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 $\{\bar{x} \mid A\bar{x} \leq \bar{c}\}$. Currently, our tool has the following functionalities:

- Computation of optimal strictly admissible integer lattice by exhaustive search with the approach suggested in [7].
- Computation of the successive minima and of a set of corresponding minimum vectors.
- Computation of the gauge functions $F_i$ and $F_i^n$.
- Implementation of the generalized basis reduction [15].
- Implementation of the different heuristics of [6].

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

### 4.1 Rogers’ Heuristic

The first heuristic proposed in [6, 7] to get a strictly admissible integer lattice is an adaptation of a mechanism due to Rogers [12], for a $0$-symmetric polytope $K$ in dimension $n$.

**Heuristic 1.**

- Choose $n$ positive integers $\rho_i$, s.t. $\rho_i$ is a multiple of $\rho_{i+1}$, and $\dim(L_{i+1}) \leq i - 1$, where $L_i = \text{Vect}(K/\rho_i \cap \mathbb{Z}^n)$.
- Choose a basis $(\bar{a}_i)_{1 \leq i \leq n}$ of $\mathbb{Z}^n$ s.t. $L_i \subseteq \text{Vect}(\bar{a}_1, \ldots, \bar{a}_{i-1})$.
- Define $\Lambda$ the lattice generated by the vectors $((\rho_i \bar{a}_i)_{1 \leq i \leq n}$.

The correctness and worst-case quality of such a strategy are analyzed in [6, 7]. It gives a correct mapping when $\rho_1 \lambda_i(K) > 1$ with $\lambda_i(K)$ the $i$-th successive minimum of $K$: $\lambda_i(K) = \min\{\lambda > 0 \mid \dim(\text{Vect}(\lambda K \cap \mathbb{Z}^n)) \geq i\}$

We now explain how we compute the scaling factors $\rho_i$ and the basis $(\bar{a}_i)_{1 \leq i \leq n}$. 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 $p$ 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 [19].

#### 4.1.1 Computing the Successive Minima

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

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

The constraint $\bar{x} \neq 0$ makes this system not linear. But, if there exists such an $\bar{x}$, then $x_i \geq 1$ or $x_i \leq -1$ for some $i$. Moreover, as $K$ is $0$-symmetric, if $\bar{x} \in K$, then $-\bar{x} \in K$, so we can look only for $\bar{x}$ 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:

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

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

**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 $N \geq 3$):

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

see Figure 2.(b). The first minimum of $K$ is $1/(N - 1)$, obtained when looking for a point $\bar{x}$ 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 $\lambda_1(K), \ldots, \lambda_{n-1}(K)$ have been computed, with their corresponding $\bar{x}_1, \ldots, \bar{x}_{n-1}$. The $i$-th minimum is the smallest $\lambda$ such that $K/\lambda$ contains an integer point $\bar{x}$, linearly independent with $\bar{x}_1, \ldots, \bar{x}_{i-1}$. To express this condition, we first compute a basis of $\mathbb{Z}^n$ whose first $(i - 1)$ vectors span the same space as $\bar{x}_1, \ldots, \bar{x}_{i-1}$. For that, we define the $n \times (i - 1)$ matrix $X$ with columns $\bar{x}_1, \ldots, \bar{x}_{i-1}$, and we compute the Hermite form [16] of $X$. $X = \bar{Q}H$ where $\bar{Q}$ 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 $\bar{x} = \bar{Q}\bar{y}$. In this representation, $\bar{x}$ is linearly independent with $\bar{x}_1, \ldots, \bar{x}_{i-1}$ if and only if $y_i \geq 1$ or $y_j \leq -1$ for some $j \geq i$. Again because $K$ is $0$-symmetric, we have:

$$\lambda_i(K) = \min_{1 \leq i \leq n} \min\{\lambda \in \mathbb{Q} \mid \exists \bar{y} \in \mathbb{Z}^n, y_i \geq 1, A\bar{Q}\bar{y} \leq \lambda \bar{c}\}$$

Solving this system, we get $\lambda_i(K)$, a corresponding solution $\bar{y}_i$, and, finally, $\bar{x}_i = \bar{Q}^{-1}\bar{y}_i$. 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 $\bar{x}_i$’s as these are not uniquely defined.

Our first implementation uses the public-domain tool GLPK [11] 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 $\bar{x}_i$ 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 $A\bar{x}_i \leq \lambda \bar{c}$. We thus consider each equation of this system to build $\lambda_i(K)$ as the quotient of two integers.

Our second implementation uses PIP [9, 17], a parametric linear programming solver, which gives the lexicographic minimum in a parameterized polyhedron $\{A\bar{x} \leq \bar{b} \bar{t} + \bar{c}\}$ where $\bar{x}$ represents nonnegative unknowns and $\bar{t}$ represents integer parameters. Using PIP is a bit tricky as 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 use PIP for mixed integer linear programming. For that, as explained in [9], the first step is to consider that integer unknowns ($\bar{x}$ in our system) are parameters and find, with parametric (rational) linear programming, the minimum of $\lambda$ as a function of $\bar{x}$. The
solution is a set of clauses, where each clause $u$ defines a solution $\lambda^u$ as an affine expression $f^u(x)$ of $x$, valid when $x$ belongs to some polyhedron $P^u$. The second step finds, for each clause $u$, thanks to integer linear programming, the minimum of $f^u(x)$ over all integer elements $x$ of $P^u$. The minimum of these minima gives the desired minimum $\lambda$.

**Main example (cont’d)** Suppose $N = 100$ for example. Our GLPK implementation first finds $\lambda_1(K) = 0.010101 \ldots$ with $x_1 = (1, 0)$, and $\lambda_1(K)$ is recomputed as $1/99$. Then, it finds $\lambda_2(K) = 1$ with $x_2 = (0, 1)$. Our PIP implementation finds $\lambda_1(K)$ directly as the fraction $1/99$ for $x_1 = (1, 0)$. Then, as PIP looks for the lexicographic minimum, it finds $x_2 = (-98, 1)$, $\lambda_2(K) = 1$. Thus, the $x_i$ may not be the same with GLPK or PIP, while the $\lambda_i$ 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. For simplicity, 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 [7] to get modular mappings 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 $x_i$) even if $K$ is linearly parametrized as $\{Ax \leq b + \epsilon \}$. Unfortunately, this does not work as $\lambda$ is then a multiplicative factor of the parameters $n$ in the linear programs that need to be solved. Looking for $1/\lambda$ instead of $\lambda$ leads to the same difficulty as $1/\lambda$ is now a multiplicative factor of $x$. Also, even if we were able to compute $\lambda_i(K)$, our technique to make sure that $x_2$ and $x_1$ are linearly independent would fail if $x_1$ is parameterized too, unless we can complete $x_1$ into a basis in a parametric way. So, this seems difficult. But $\lambda_i(nK) = \lambda_i(K)/n$, 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 function 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 $(\bar{a}_i)_{1 \leq i \leq n}$

Once we get the successive minima, we can find the scaling factors $\rho_i$ and an adequate basis $(\bar{a}_i)_{1 \leq i \leq n}$ of $\mathbb{Z}^n$, as follows. For the vectors $(\bar{a}_i)_{1 \leq i \leq n}$, a possible choice, suggested in [6], is to use the vectors $\bar{a}_i$ associated with the successive minima $\lambda_i(K)$. Indeed, if $\rho_i \lambda_i(K) > 1$, then the vector space $L_i = \text{Vect}(K/\rho_i \cap \mathbb{Z}^n)$ has at most $(i-1)$ dimensions. Furthermore, $L_i \subseteq \text{Vect}(\bar{x}_i, \ldots, \bar{x}_{i-1})$. However, the vectors $(\bar{x}_i)_{1 \leq i \leq n}$, 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 $X = QA^t$ and we define the vectors $\bar{a}_i$ as the columns of $Q$.

For the scaling factors $\rho_i$, a possibility is to choose $\rho_i$ to be the smallest power of 2 strictly larger than $1/\lambda_i(K)$. It turns out however that the divisibility condition for the $\rho_i$, 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 $\rho_i$ 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 $(\rho_i \bar{a}_i)_{1 \leq i \leq n}$ is strictly admissible for $K = \{Ax \leq \bar{c}\}$. This amounts to check that there is no $\bar{x} \neq 0$ such that $\bar{x} = \sum_{i=1}^{n} \rho_i \bar{a}_i$ and $Ax \leq \bar{c}$. For each $x_i$ an integer. This can be done by solving $n$ linear programs, using the same technique we used to compute $\lambda_1$.

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

**Main example (cont’d)** Here, 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 $x_1 = (1, 0)$ and $x_2 = (0, 1)$. With Heuristic 1, we get $\rho_1 = 128$ and $\rho_2 = 2$ with the mapping $\sigma(i, j) = (i \mod 128, 12j \mod 2)$, which can be simplified into $\sigma(i, j) = (i \mod 12, 1 \mod 2)$ as $0 \leq i < N \leq 128$. 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 efficient. Going to the next power of 2 as in Heuristic 1 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 $x_i$, its components in absolute value. One can also use, after all $x_i$ are computed, a form of basis reduction to get a simpler basis. One example is the generalized basis reduction of Lovász and Scarf [15] 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 $L_i$, one can limit the modifications to “upper triangular” changes, i.e., simplifying $\bar{x}_i$ with linear combinations of the previous $x_j$, $j < i$. 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 $\sigma(i, j) = (i + 98j \mod 100, 12j \mod 2)$ with Heuristic 1.a. It can be simplified into $\sigma(i, j) = (i - 2j \mod 100, 12 \mod 2)$, but, still, this is more complicated than the mapping found with GLPK. To get $x_2 = (0, 1)$ as for GLPK instead of $x_2 = (-98, 1)$, we need to complete our implementation so that it simplifies the basis. Looking for $x_2$ with minimal components in absolute value would lead to $x_2 = (0, 1)$. One can also simplify $\bar{x}_2$ with $\bar{x}_1$, $L_2$ 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, we get $x_1 = 128$ and $x_2 = (0, 1)$ with the mapping $\sigma(i, j) = (i \mod 128, 12j \mod 2)$, which can be simplified into $\sigma(i, j) = (i, 12j \mod 2)$ as $0 \leq i < N \leq 128$. 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 efficient. Going to the next power of 2 as in Heuristic 1 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 $x_i$, its components in absolute value. One can also use, after all $x_i$ are computed, a form of basis reduction to get a simpler basis. One example is the generalized basis reduction of Lovász and Scarf [15] 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 $L_i$, one can limit the modifications to “upper triangular” changes, i.e., simplifying $\bar{x}_i$ with linear combinations of the previous $x_j$, $j < i$. But how to derive the “simplest” mapping for a given lattice is still an issue.
variant of Heuristic 1 explained in the previous section, is, on average (but not always), the one that leads to the smallest lattice. 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 \(\rho_i\) 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 [6, 7] as generalizations of the heuristic of Lefebvre and Feautrier [14]. They lead to memory allocations of the form \(\sigma(i) = Mc \mod b\) where \(b\) 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 [6, 7].

The function \(F(z) = \min\{\lambda > 0 \mid z \in \lambda K\}\) defines a norm such that \(F(\alpha z) = |\alpha| F(z)\), called the gauge function of \(K\). Given some vectors \((\vec{a}_i)_{1 \leq i \leq s}\), one can define \(F_i(z) = \min\{F(y) \mid y \in \vec{a}_i + \langle \vec{a}_1, \ldots, \vec{a}_{i-1} \rangle\}\), 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 \(F_1(\vec{a}_i) = \min\{F(y) \mid y \in \vec{a}_i + \langle \vec{a}_1, \ldots, \vec{a}_{i-1} \rangle\}\), for \(1 \leq i \leq n\).
- Choose \(n\) integers \(\rho_i\) such that \(\rho_1 F_1(\vec{a}_1) > 1\).
- Define \(L\) the lattice generated by the vectors \((\rho_1 \vec{a}_1, \ldots, \rho_{n} \vec{a}_{n})\).

Again, we refer to [6, 7] 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:

\[
F_i(\vec{a}_i) = \min\{F(y) \mid y \in \vec{a}_i + \langle \vec{a}_1, \ldots, \vec{a}_{i-1} \rangle\} = \min\{\lambda > 0 \mid y \in \lambda K, y = \vec{a}_i + \sum_{j=1}^{i-1} y_j \vec{a}_j\} = \min\{\lambda > 0 \mid A\vec{y} \leq \lambda \vec{c}, \vec{y} = \vec{a}_i + \sum_{j=1}^{i-1} y_j \vec{a}_j\}
\]

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

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

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

\[
F_i(\vec{a}_i) = \min\{1/\rho > 0 \mid \vec{z} \in K, \vec{z} = \rho \vec{a}_i + \sum_{j=1}^{i-1} z_j \vec{a}_j\} = 1/\max\{\rho > 0 \mid A\vec{z} \leq B\vec{u} + \vec{c}, \vec{z} = \rho \vec{a}_i + \sum_{j=1}^{i-1} z_j \vec{a}_j\}
\]

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 \(F_i(\vec{a}_i)\)s are computed, we let \(\rho_i = [1/F_i(\vec{a}_i)] + 1\). If \((\vec{a}_i)_{1 \leq i \leq n}\) defines a basis of \(\mathbb{Z}^n\), again, the mapping is \(\sigma(i) = Mi \mod b\) where \(M\) is the inverse of the matrix whose columns are the \(\vec{a}_i\) and \(b\) 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 \(1/F_1(\vec{a}_1) = N = 1 - 1\) and \(\rho_1 = N\), then \(F_2(\vec{a}_2) = 2 = 2\) and \(\rho_2 = 2\), with the mapping \(\sigma(i_1, i_2) = (i_1 \mod N, i_2 \mod 2)\). Also, as mentioned in [6], with a slight basis change, we can also obtain a valid 1D mapping as \(\sigma(i_1, i_2) = 2i_1 + i_2 \mod 2N\). For this particular case, we are not far from the optimal \(\sigma(i_1, i_2) = 2i_1 \mod 2N - 3\). Picking the vectors in the opposite order leads to \(\rho_1 = 2, \rho_2 = N\) and the mapping \((i_2 \mod 2, i_1 \mod N)\). We can also get the valid 1D mapping \(N/2 + i_1 \mod 2N\).

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 connection with Heuristic 2 is explained in [6]. 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 \(K\), i.e., \(K^* = \{\vec{y} \mid \vec{x}^T \vec{y} \leq 1\ for all \vec{x} \in K\}\). For the implementation, we use the alternative definition of \(F_i^*(\vec{c}_i)\) as \(F_i^*(\vec{c}_i) = \sup\{\vec{c}_i, \vec{x} \mid \vec{x} \in K, \vec{c}_i^T \vec{x} = 0, \forall j < i\}\).

**Heuristic 3.**
- Choose \(n\) linearly independent integral vectors \((\vec{c}_1, \ldots, \vec{c}_n)\).
- Compute \(F_i^*(\vec{c}_i) = \sup\{\vec{c}_i, \vec{x} \mid \vec{x} \in K, \vec{c}_i^T \vec{x} = 0, \forall j < i\}, 1 \leq i \leq n\).
- Choose \(n\) integers \(\rho_i\) 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 \(b\) the vector such that \(b_i = \rho_i\).

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 [6] 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 \((\vec{a}_i)_{1 \leq i \leq n}\) and \((\vec{c}_i)_{1 \leq i \leq n}\), 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 [8]), 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 [6, 7] (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 [20], Polylib and PIP [17]. 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 [22]. We use ROSE to extract iteration domains \(I_S\) 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
### Table

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Array Identifier</th>
<th>Storage Mapping found</th>
<th>Method</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mapping</td>
<td>Compressed</td>
<td>H1</td>
<td>H2</td>
</tr>
<tr>
<td>durbin.c</td>
<td>i → i mod 1</td>
<td>1</td>
<td>⊗ × × ×</td>
<td>0.1</td>
</tr>
<tr>
<td>beta</td>
<td>i → i mod 1</td>
<td>1</td>
<td>⊗ × × ×</td>
<td>0.1</td>
</tr>
<tr>
<td>sum</td>
<td>(i, j) → (i mod 1, j mod 1)</td>
<td>1</td>
<td>× × × ×</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>(i, j) → (i mod 100, j mod 2)</td>
<td>200</td>
<td>× × × ×</td>
<td>1.8</td>
</tr>
<tr>
<td></td>
<td>(i, j) → (i mod 1, 2i + j mod 197)</td>
<td>197</td>
<td>×</td>
<td>17</td>
</tr>
<tr>
<td>reg_detect.c</td>
<td>sum → (i, j) → (i mod 6, j mod 6)</td>
<td>36</td>
<td>× × ×</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j) → (i mod 3, i + j mod 9)</td>
<td>27</td>
<td>× × ×</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j) → (i mod 6, j mod 6)</td>
<td>36</td>
<td>× × ×</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j, k) → (i mod 6, j mod 6, k mod 64)</td>
<td>2304</td>
<td>× × ×</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j, k) → (i mod 3, i + j mod 9, k mod 64)</td>
<td>1728</td>
<td>× × × ×</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>4</td>
<td>× × ×</td>
<td>0.7</td>
</tr>
<tr>
<td>dynprog.c</td>
<td>c → (i, j) → (i mod 9, j mod 9)</td>
<td>81</td>
<td>× × ×</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>sum → (i, j) → (i mod 1, 13i + j mod 61)</td>
<td>61</td>
<td>× × ×</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>1</td>
<td>× × ×</td>
<td>6.1</td>
</tr>
<tr>
<td>gauss.c</td>
<td>acc1</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>1</td>
<td>× × ×</td>
</tr>
<tr>
<td></td>
<td>acc2</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 1)</td>
<td>1</td>
<td>× × ×</td>
</tr>
<tr>
<td></td>
<td>gImp</td>
<td>(i, j, k) → (i mod 50, j mod 48)</td>
<td>2400</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>(i, j, k) → (i mod 48, j mod 50)</td>
<td>2400</td>
<td>× × ×</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>(i, j, k) → (j + i mod 2, 2j + 5i mod 1200)</td>
<td>2400</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>mod_detect_kern.c</td>
<td>Delta</td>
<td>(i, j, k) → (k mod 10, j mod 1, i mod 1)</td>
<td>10</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>ODelta</td>
<td>(i, j, k) → (i mod 1, j mod 1, k mod 10)</td>
<td>10</td>
<td>× × ×</td>
</tr>
</tbody>
</table>

### Figure 3

Experimental results for the kernels given in Figure 4. 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.

polyhedral operations (union, intersection, projection, and convex hull) required by the different steps of our method are computed thanks to the polyhedral library Polylib [17]. Finally, the lexicographic minima/maxima of integral polytopes are derived thanks to the PIP library [9, 17].

We applied our tools to contract temporary arrays of several image processing kernels [32] 1. We chose these kernels as they are used in all related papers and because they are short enough to be given here (see Figure 4). Figure 3 gives the results of our experiments. The benchmarks gather (1) column gives the array size after remapping, i.e., the products are derived thanks to the PIP library [9, 17].

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

---

1 Many thanks to Florin Balasa who gave us these code examples.
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 ∼ 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) | 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) ↦ (i + j mod 3m, j mod m) if N = 2m and (i, j) ↦ (i - (3m + 2j) mod (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 x ≤ b}. For the array mean, we get the lattice generated by (3, 0) and (1, -1), with determinant 3. Multiplying by λ, we get that Aλ is strictly admissible for {A x ≤ Aλ}. For the array mean and λ = 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 x ≤ B}, it is weakly admissible (i.e., boundary intersection is acceptable) for {A x ≤ B + 1}, which is here {A x ≤ 2B}. Multiplying by m, we get that mA is weakly admissible for {A x ≤ 2mB} and thus strictly admissible for {A x < Nm}. We retrieve the optimal mapping for N = 2m with size 3m^2. The same mapping is valid for N = 2m - 1 with size ∼ 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 [7], 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 more general programs by providing a conservative lifetime analysis (containing the real lifetime). Our lifetime analysis is already 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 [7] can impact on spatial data locality, it would be good to have a method to select the best one.

References


# Kernel Implementation

```c
#define M 50
#include "gauss.h"

main() {
    int i, j, k;
    int sum_t[M][M], mean[M][M];
    int diff[M][P][M], sum_d[M][P][M];
    int tangent[M];
    int path[M][M];
    int i, j, k;
    for (i=0; i<=N-2; i++) {
        for (j=i+1; j<=N-1; j++) {
            sum_d[i][j][k] = sum_d[i][j][k-1] + path[j][k-1] + mean[j][k];
            if (j<i+n-1) {
                sum_d[i][j][k] = sum_d[i][j][k] + tangent[i][j];
            } else if (j<i+n-1) {
                sum_d[i][j][k] = sum_d[i][j][k] + W[i][k];
            }
        }
    }
}
```

# Reference


Figure 4. Source code of kernels.