Laboratoire de l'Informatique du Parallélisme

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

Lattice-Based Memory Allocation

Alain Darte, Rob Schreiber,
and Gilles Villard

April 2004

Research Report N° 2004-23

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`

CENTRE NATIONAL
DE LA RECHERCHE
SCIENTIFIQUE

INRIA

Lattice-Based Memory Allocation

Alain Darte, Rob Schreiber, and Gilles Villard

April 2004

Abstract

We investigate the technique of storing multiple array elements in the same memory cell, with the goal of reducing the amount of memory used by an array variable. This reduction is both important and achievable during the synthesis of a dedicated processor or code generation for an architecture with a software-controlled scratchpad memory. In the former case, a smaller, less expensive circuit results; in the latter, scratchpad space is saved for other uses, other arrays most likely. The key idea is that once a schedule of operations has been determined, the schedule of references to a given location is known, and elements with disjoint lifetimes may share a single memory cell, in principle. The difficult problem is one of code generation: how does one generate memory addresses in a simple way, so as to achieve a nearly best possible reuse of memory? Previous approaches to memory reuse for arrays consider some particular affine (with modulo expressions) mapping of indices, representing the data to be stored, to memory addresses. We generalize the idea, and develop a mathematical framework based on critical integer lattices that subsumes all previous approaches and gives new insights into the problem. We place the problem in a broader mathematical context, showing its relation to real critical lattices, successive minima, and lattice basis reduction; finally, we propose and analyze various strategies for lattice-based memory allocation.

Keywords: Program transformation, memory allocation, memory size reduction, admissible lattice, critical determinant, successive minima.

Résumé

Nous explorons le problème de la réutilisation de la mémoire, dans le but de réduire la taille nécessaire pour une variable de type tableau, dans le contexte de la synthèse de processeurs dédiés ou de la compilation vers des architectures possédant des mémoires programmables de type “scratchpad”. Dans le premier cas, le résultat est un circuit plus petit, moins coûteux, dans le second cas, de l'espace mémoire est économisé permettant éventuellement le stockage d'autres tableaux. L'idée est qu'une fois l'ordonnancement des calculs défini, l'ordre des accès à une adresse mémoire donnée est connu, et en principe, des éléments de périodes de vie disjointes peuvent partager une même cellule mémoire. Le problème est celui de la génération de code : comment générer les adresses mémoire pour obtenir le meilleur (ou assez bon) partage de la mémoire ? Toutes les approches précédentes considèrent des fonctions d'adressage particulières, affines avec des opérations modulo, associant à chaque indice représentant une donnée à sauvegarder une adresse mémoire. Nous développons un modèle mathématique basé sur les réseaux critiques entiers qui nous permet de généraliser toutes ces approches et qui offre de nouvelles vues sur le problème. Notre modélisation montre le lien avec les réseaux critiques, les minima successifs et les réductions de base, et nous permet d'analyser différentes stratégies pour les allocations mémoires construites à partir de réseaux.

Mots-clés: Optimisations de programmes, de mémoire, réseau admissible, minima successifs.

Contents

Lattice-Based Memory Allocation

Alain Darte
CNRS, LIP, ENS Lyon
46 Allée d'Italie,
69364 Lyon Cedex 07, France
Alain.Darte@ens-lyon.fr

Rob Schreiber
Hewlet Packard Laboratories,
1501 Page Mill Road,
Palo Alto USA
Rob.Schreiber@hp.com

Gilles Villard
CNRS, LIP, ENS Lyon
46 Allée d'Italie,
69364 Lyon Cedex 07, France
Gilles.Villard@ens-lyon.fr

Abstract

We investigate the technique of storing multiple array elements in the same memory cell, with the goal of reducing the amount of memory used by an array variable. This reduction is both important and achievable during the synthesis of a dedicated processor or code generation for an architecture with a software-controlled scratchpad memory. In the former case, a smaller, less expensive circuit results; in the latter, scratchpad space is saved for other uses, other arrays most likely.

The key idea is that once a schedule of operations has been determined, the schedule of references to a given location is known, and elements with disjoint lifetimes may share a single memory cell, in principle. The difficult problem is one of code generation: how does one generate memory addresses in a simple way, so as to achieve a nearly best possible reuse of memory?

Previous approaches to memory reuse for arrays consider some particular affine (with modulo expressions) mapping of indices, representing the data to be stored, to memory addresses. We generalize the idea, and develop a mathematical framework based on critical integer lattices that subsumes all previous approaches and gives new insights into the problem. We place the problem in a broader mathematical context, showing its relation to real critical lattices, successive minima, and lattice basis reduction; finally, we propose and analyze various strategies for lattice-based memory allocation.

Keywords Program transformation, memory allocation, memory size reduction, admissible lattice, critical determinant, successive minima.

1 Introduction

The scheduling and mapping of computation are major problems faced by compilers and parallelizers for parallel machines or embedded systems, and compilers for automatic synthesis of hardware accelerators. In automatic hardware synthesis, a particularly important problem is the design of the buffers needed to store data in an application-specific circuit or between two such communicating circuits. Previously developed systems that synthesize hardware accelerators from high-level programs have faced this issue. Salient examples are the HP Labs PICO project [17, 7] (now pursued in the start-up Synfora [30]), the IMEC Atomium project [3, 9, 34], the Compaan project [18, 35] at Leiden, and the Alpha project [27, 26] in Rennes. As noted in [33], there is as yet no theoretical framework to describe and analyze the interplay between scheduling (i.e., *when* statements are executed) and storage required (i.e., *where* results are stored). There is now a mature theory for the scheduling problem (see for example [8] for a survey), where parallelism is to be maximized and both communication cost and storage requirement are ignored; this is not the case, however, for the storage allocation problem, even when the schedule of loop iterations is given. While several heuristics have been proposed to reduce the amount of memory needed by a given scheduled program, there has been no mathematical framework available in which to compare heuristics and analyze them with respect to some optimal theoretical solution.

Like those who have come before us to this work, we define a **mapping** of indices to memory locations (also called an **allocation** or **layout**) for the intermediate and final results of the program, and we *reuse memory locations* by using a layout that stores several elements of an array in the same location. To do so does not violate program semantics if, with the schedule given, the array elements that share a memory location never simultaneously hold live values.

Earlier approaches were, briefly, as follows. De Greef, Catthoor, and De Man [9] proposed to layout arrays by selecting a canonical linearization (such as column-major or row-major) followed by a wrapping with a modulo operation. For example, for a 2-dimensional square array of size N, they consider a mapping of the form $Ni+j \bmod b$ for the array element (i, j); the storage required is b and this is legal for some large enough integer $b \leq N^2$ (see details in Section 2.1). Among the $2^n n!$ possible canonical linearizations of an n-dimensional array, they pick the one yielding the smallest value for b. Lefebvre and Feautrier [21] proposed a more general technique based on the original loop indices (rather than the array indices), with a wrapping thanks to a modulo operation in each dimension. For a 2-dimensional loop, the result of statement S in iteration (i, j) is stored in a new array A_S, and mapped to memory location $A_S(i \bmod b_i, j \bmod b_j)$ where b_i and b_j are well-chosen integers (see details in Section 2.2). Quilleré and Rajopadhye [26] also use a dedicated array for the results of a statement, but their access function is a "projection" along some well-chosen (non-parameterized) axes that depend on the program schedule (the details are in Section 2.3) and are not necessarily aligned with the original loop indices. Clearly, any two statement instances whose index vectors differ by an element of the null space of the projection are mapped to the same array element. They mention the use of modulo operations, but only with a brief analysis. They showed that, under some hypotheses (mainly that iteration domains have large sizes in all dimensions), their strategy leads to a projection onto a subspace of smallest dimension; that is, they choose a projection of maximum nullity. More recently, Thies, Vivien, Sheldon, and Amarasinghe [33], extending the work of Strout, Carter, Ferrante, and Simon [29], considered the problem of reusing storage along only one axis (i.e., with a modulo operation in a single dimension) and how to find such a good axis. Although part of their technique is more limited than previous work, they also consider memory reductions valid for *any* schedule that respects the dependences of the program.

All of these previous techniques are characterized by the use of specializations of the (linear) **modular mappings** we study in this paper, i.e., memory mappings that access multi-dimensional arrays with affine functions followed by a modulo operation in each dimension. Thies, Vivien, Sheldon, and Amarasinghe, aware of the limitations of their work due to the use of a modulo operation in a single dimension, conclude by mentioning the need for a technique able to consider the *"perfect storage mapping* [that] *would allow variations in the number of array dimensions, while still capturing the directional and modular reuse of the occupancy vector and having an efficient implementation"*. This is exactly our goal here; to propose and study a mathematical framework that allows us to capture all linear modular storage allocations, to define optimality, and to discuss the quality of heuristics with respect to the optimal mapping.

To introduce and motivate such a framework, we describe in more details, the heuristic algorithms of De Greef, Catthoor, and De Man (Section 2.1), of Lefebvre and Feautrier (Section 2.2), and of Quilleré and Rajopadhye (Section 2.3). Through examples, we identify their main underlying concepts and illustrate their limitations. In Section 3, we model the problem of memory reuse in a scheduled program. We define the type of memory allocation that we target, linear **modular** allocations, and identify the constraints they must satisfy to respect the semantics of the program. These constraints are described by a set of **conflicting indices**, which corresponds to pairs of elements that may not share the same memory location.

Our study relies on some key underlying mathematical facts (that we derive in Section 4). These relate to a correspondence between modular allocations and integer **lattices**; a further correspondence between the validity of a modular allocation for a given program and the **strict admissibility** of the related lattice for the set of differences of conflicting indices; and finally, exact equality of the number of memory locations used by a modular allocation and the determinant of the corresponding lattice. Our problem is equivalent, then, to finding the integer lattice of minimum determinant that is strictly admissible for the difference set of the conflicting indices. This equivalence allows us to define optimality for modular allocations, to get lower and upper bounds, to develop approximation heuristics, and to analyze the performance of previous heuristics. Most of these results are obtained assuming that the set of differences of conflicting indices is represented or approximated as the set of integer points in a full-dimensional 0-symmetric polytope K. Thanks to this practical assumption, we can give a *geometric view* of the problem, derive upper and lower bounds linked to the *volume* of K, and study *algorithmic mechanisms* to approach these bounds, using techniques from the theory of successive minima and reduction theory. In particular, we show that any valid modular allocation requires at least $\mathrm{Vol}(K)/2^n$ memory locations (Section 4) and that the heuristics we develop (Section 5) provide an allocation that needs at most $c_n \mathrm{Vol}(K)$ locations (where c_n depends on the dimension n only). Our heuristics are thus optimal up to a multiplicative factor. We also show how to determine an allocation in an array of smallest dimension (Theorem 1), and we explain how to get a one-dimensional linear mapping (i.e., with a single modulo operation) with guaranteed performance (Corollaries 1 and 2).

In Section 6, we come back to scheduled programs and identify some particular cases for which the heuristics get simpler, while still achieving provably good performance. As a particular case of our heuristics, we retrieve the mechanism of Lefebvre and Feautrier, which, as we show, should be applied in a well-chosen basis, not always the basis given by the original loop indices. In Section 7, we present a case study of an interesting application, the DCT computation in JPEG and MPEG codecs, which illustrates the concepts and techniques we discussed. We conclude in Section 8.

2 Previous work on memory reuse for scheduled programs

In this section, we describe in more detail the results discussed in Section 1, highlighting their underlying concepts and illustrating, by example, their limitations. We want to understand when, and why, these approaches work or fail, and to derive better solutions and handle more general cases. We aren't concerned, here, with the general problem of compilation of loops, including, among other things, the issues of how to analyze a program, how to represent it, and how loops can be scheduled while respecting dependences. We instead keep the discussion as general as possible, so as to develop a framework suitable for memory allocation whatever the program analysis and program representation.

We first recall some definitions and give some notations that we use to describe the different approaches to the problem. We use the classical representation of loops (possibly not perfectly nested) with iteration vectors. An n-dimensional iteration vector $\vec{i} = (i_1, \ldots, i_n)$ represents the values of the counters of n nested loops. Each statement S has an associated iteration domain $\mathcal{I}_S \subset \mathbb{Z}^n$, such that $\forall \vec{i} \in \mathcal{I}_S$, there is a dynamic instance (or operation) $u = (S, \vec{i})$ of statement S at iteration $\vec{i}$ during the execution of the program. We assume that a scheduling function θ (that may or may not express some parallelism) is given, which assigns to each operation u a "virtual" execution time, i.e., an element of a partially ordered set $(\mathcal{T}, \preceq)$. The operation u will be computed strictly before the operation v iff $\theta(u) \prec \theta(v)$. A schedule respects dependences, especially flow dependences, i.e., if v uses a value computed by u, then $\theta(u) \prec \theta(v)$. Traditionally, $(\mathcal{T}, \preceq)$ is the set of integers with the order $\leq$ (and the schedule is called a **one-dimensional schedule**), or a discrete d-dimensional space with the lexicographic order (and the schedule is called a **multi-dimensional schedule** of dimension d or a d-dimensional schedule), and the mapping θ, when restricted to operations $u = (S, \vec{i})$, is an affine function of $\vec{i}$; but these restrictions are not needed here.

Throughout the paper, we use two very simple examples. For each technique, we use the examples to show how the constraints for memory reuse are represented, what type of memory mappings are considered, what are the different concepts used to derive the mapping, and what are the different traps into which the technique can fall.

Example 1 Consider the code fragment in Figure 1 and suppose that we can reuse the memory for A, i.e., A is not "live-out" from the code fragment. Suppose that the two loops are scheduled sequentially as they are written but that, for some reason, the second loop is pipelined, with respect to the first one, one "clock cycle" later. In other words, $\theta(S,(i,j)) = (i,j)$ and $\theta(T,(i,j)) = (i,j+1)$ if $0 \leq j < N-1$, $\theta(T,(i,j)) = (i+1,0)$ if $j = N-1$. For this particular case, we can also describe the schedule with a one-dimensional schedule $\theta(S,(i,j)) = Ni + j$ and $\theta(T,(i,j)) = Ni + j + 1$. This kind of pipelined schedule occurs typically when compiling communicating processes, the values of A being placed in an intermediate buffer whose size is to be optimized.

```
DO i = 0, N − 1                DO i = 0, N − 1
  DO j = 0, N − 1                DO j = 0, N − 1
    S: A(i, j) = ...               T: B(i, j) = A(i, j) + ...
  ENDDO                          ENDDO
ENDDO                          ENDDO
```

Figure 1: Code for Example 1, the second loop starts 1 clock cycles later.

If we consider that a value is **dead** only at the end of the read in T, then each array element is **live** during two cycles and only two values are live at any one time. This corresponds to the classical quantity MAXLIVE used in register allocation, which is the maximal number of values simultaneously live, here equal to 2. Therefore, we can use two memory cells. The mapping $A(i,j) \mapsto Ni+j \bmod 2$ (plus a base address in memory) is a valid mapping that requires only two memory cells; note that if N is even this amounts to the mapping $j \bmod 2$, otherwise to $(i+j) \bmod 2$. How can a compiler automatically find such a mapping?

As pictures are much easier to digest than symbols and logic, we will adopt a pictorial convention for our examples, showing iteration spaces in which one typical iteration $\vec{i}$ is singled out and shown in black. In grey we show the set of iterations that store new values while the value stored at $\vec{i}$ is still live. The grey iterations may not write to the same location as the black. If $\vec{j}$ is a grey iteration, we say that $\vec{i}$ and $\vec{j}$ **conflict** and we denote this by $\vec{i} \bowtie \vec{j}$ (we define this more formally in Section 3.1). For example, in Figure 2, we depict the two generic cases. For (i,j) with $j < N-1$, S at iteration $(i, j+1)$ may not write to the same location as S at iteration (i,j). Similarly, S at iteration $(i+1, 0)$ may not write to the same location as S at iteration $(i, N-1)$.

In this example, the iteration space of S and the index space of the array A are aligned (i.e., S at iteration (i,j) writes to A at location (i,j)), therefore Figure 2 can be interpreted in a different way. For a given array element $A(i,j)$ (in black), the other points (in grey) represent the array

Figure 2: Two different cases of conflicting iterations for the code of Example 1, for $N = 9$.

elements that may not be mapped to the same memory address as $A(i, j)$. (To avoid confusion, we point out however that, since i is the horizontal axis in Figure 2, a row of the array spans a column in the figure. A foolish consistency, we feel, ought not encumber us.) □

Example 2 Consider the code fragment in Figure 3. For each iteration, statement S writes an element of A that is read for the last time in the next iteration of the outermost loop, and is read there only after the write by statement S. The set of conflicting iterations for this example is shown in Figure 4. □

```
DO i = 0, N − 1
  DO j = 0, N − 1
    S: A(i, j) = . . .
    T: B(i, j) = A(i − 1, j) + A(i, j)
  ENDDO
ENDDO
```

Figure 3: Code for Example 2.

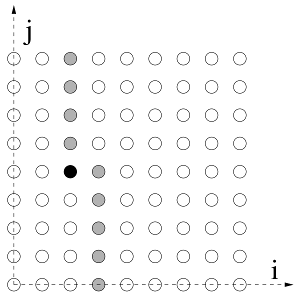

Figure 4: Example of conflicting iterations for $N = 9$.

2.1 De Greef, Catthoor, and De Man

In [9] (and earlier works), De Greef, Catthoor, and De Man, working on the Atomium project [3], identified the need for memory reduction techniques for embedded multimedia applications. Their method is based on an analysis of the array-element addresses used in the program. They introduce two techniques, an inter-array storage optimization, which refers to the relative position of different arrays in memory, and an intra-array storage optimization, which refers to the internal organization of an array in memory. Let us describe the general idea of the second optimization, the intra-array storage optimization, since this is the technique we want to analyze.

First, the schedule of the program (i.e., the ordering of computations in the program) is *linearized* in some way (note that this is not always possible), assigning to each operation u an integer virtual time $\theta(u)$. Then, for each array accessed in the program, the following is done. First, a **canonical linearization** of the array is chosen. For an n-dimensional array, they consider $2^n n!$ canonical linearizations, $n!$ choices for the order in which dimensions are considered and, for each order, 2^n choices depending whether each dimension is traversed backwards or forwards. For example, for a $2D$ array A of size (M, N), there are eight canonical linearizations: $A(i, j)$ can be mapped in memory to a base address (which we will ignore in the rest of this paper) plus $Ni + j$, $Ni - j$, $-Ni+j$, $-Ni-j$, $i+Mj$, $-i+Mj$, $i-Mj$, or $-i-Mj$. Then, they build what they call the BOATD (Binary Occupied Address-Time Domain), which is the set of all pairs of integers (a, t) where t is a time in the program execution and a is a memory address corresponding to an element of A (in the particular array linearization being studied) that contains a value live at time t, i.e., a value that was computed before or during t and that will be read after or during t. Using the previous terminology, if (a_1, t) and (a_2, t) belong to BOATD, a_1 and a_2 conflict, i.e., $a_1 \bowtie a_2$. Then, they compute $b = 1 + \max\{a_1 - a_2 \mid (a_1, t), (a_2, t) \in \text{BOATD}\}$ and they wrap the linearization modulo b, meaning that the address of the element $A(i, j)$ in the optimized program is the base address of A plus $o \bmod b$ where o is the value given by the selected canonical linearization. This is indeed a correct mapping since b is chosen large enough so that, at any given time t, two values live at time t can never be mapped to the same location. In effect, there is a sliding *window* of b successive addresses that always covers all the live values.

Example 1 (Cont'd) Let us see how the De Greef–Catthoor–De Man technique works on this example. To make the discussion shorter, we consider only two out of the eight linearizations of A, the column-major order and the row-major order, and we assume that A is a square array of size N. We must first find, for each time $t = Ni + j$, with $0 \leq j < n$, the addresses that contain a live value at time t. These are a) the address corresponding to $A(i, j)$, written at time t, and b) the address, written at time $t - 1$, which is still live since it is read at time t. This value is $A(i, j - 1)$ for $j > 0$ and $A(i - 1, N - 1)$ for $j = 0$ (see Figure 2). For the column-major order, $A(i, j)$ is mapped to $i + Nj$, thus the BOATD contains the pairs $(i + Nj, Ni + j)$ and $(i + N(j - 1), Ni + j)$ (for $j > 0$) and the pairs (i, Ni) and $(i - 1 + N(N - 1), Ni)$. Thus, the maximal address difference for a given time t is $N(N - 1) + i - 1 - i = N(N - 1) - 1$ and the corresponding mapping is $A(i, j) \mapsto (i + Nj) \bmod N(N - 1)$, which leads only to a slight memory reduction compared to the original array. However, for the row-major order, where $A(i, j)$ is mapped to $Ni + j$, things are much better since the mapping and the time are "aligned", i.e., at time t, the written address is exactly t. Thus, at time t, two addresses are live, the address t and the address $t - 1$ (written one step earlier). In other words, the BOATD contains all pairs (t, t) and $(t - 1, t)$. Therefore, we get $b = 2$ and the mapping $A(i, j) \mapsto Ni + j \bmod 2$ is valid and requires only 2 memory cells. □

Thus, for the first example, the De Greef–Catthoor–De Man method is optimal (with a memory size equal to 2). In this case, one of the canonical linearizations of the array is identical to the iteration schedule and happens to pack all the live values into a dense interval of the address space. If, however, the array A is an array of size $M > N$ instead of N (the loop bounds), the maximal address difference for the row-major order is $Mi - (M(i - 1) + N - 1) = M - N + 1$ (for the two live values $A(i, 0)$ and $A(i - 1, N - 1)$), and we obtain a memory size equal to $b = M - N + 2$ instead of $b = 2$. Thus, considering the linearizations of the original array with respect to its full size, when only part of the array is computed, is a limitation. Also, even if all values of the array are computed, there is no guarantee that the schedule is aligned with a canonical linearization of the array, especially after possible compiler-generated loop transformations or data layout optimizations. This problem is illustrated by the following example.

Example 2 (Cont'd) It is clear that, for the code of Example 2, the best canonical allocation is the row-major order that stores conflicting indices in sequence in memory (see Figure 4 again, and remember that a "row" of the array is actually a column in the figure). In this case, the maximal difference of conflicting indices is equal to N and De Greef–Catthoor–De Man selects the mapping

$A(i,j) \mapsto Ni+j \bmod N+1$, i.e., $A(i,j) \mapsto j-i \bmod N+1$, which is optimal since there are indeed $N+1$ values simultaneously live at some point of the program.

However, suppose that, for some reason, the loop transformation $(i,j) \mapsto (i+j,j)$ is applied, i.e., the program is scheduled with the 2-dimensional schedule $\theta(i,j) = (i+j,j)$ (for both S and T). The resulting code is given in Figure 5. Considering Figure 6, which depicts a particular case of

```
DO k = 0, 2N − 2
  DO j = max(0, k − N + 1), min(N − 1, k)
    S: A(k − j, j) = . . .
    T: B(k − j, j) = A(k − j − 1, j) + A(k − j, j)
  ENDDO
ENDDO
```

Figure 5: Code for Example 2 after loop transformation $(i,j) \mapsto (i+j,j)$, with $k=i+j$.

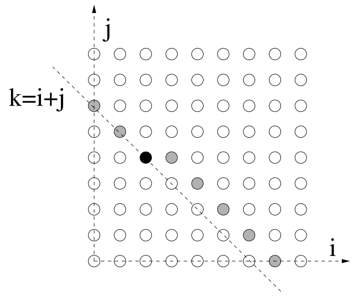

Figure 6: Example of conflicting iterations.

conflicting array elements, it is easy to see that, for any canonical linearization of A (either by row or column, backwards or forwards), the maximal address difference is of order N^2, which is far from optimal! Indeed, it is also easy to see that no two array elements in the same row (i.e., $A(i,j)$ and $A(i,j')$) are simultaneously live, so the mapping $A(i,j) \mapsto i$ is a correct mapping that requires only N memory cells. □

The Atomium group was aware of some of these limitations. To avoid reliance on a particular linearization before wrapping, Tronçon, Bruynooghe, Janssens, and Catthoor advocated use of an n-dimensional *bounding box* in the original n-dimensional space of the array indices instead of a one-dimensional window (i.e., with a single modulo operation) in the linearized space of addresses [34]. In other words, n values b_i are defined separately as the maximal index difference between indices of conflicting elements, in each dimension, so that finally all conflicting index differences belong to a multi-dimensional rectangular window. We will not discuss this approach here, since its bounding box mechanism is never better than the *successive moduli* approach of Lefebvre and Feautrier that we discuss next. Note though that a bounding box aligned with the array axes is no better for the code of Figure 5, since its size is of order N in both directions.

2.2 Lefebvre and Feautrier

Lefebvre and Feautrier [20, 21], working on automatic parallelization of static control programs, developed a technique of partial data expansion that, even if intended for other ends, can be used as a memory reduction technique. Like De Greef–Catthoor–De Man, they advocate an inter-array storage optimization (based on graph coloring); we are here concerned only with the heart of their approach, its intra-array storage optimization. Their work is strongly based on the polytope model (see [12, 4] for surveys), for dependence analysis, translation to single assignment form, scheduling and loop transformation, lifetime analysis, etc.

For storage optimization, they propose to ignore the arrays of the original program. They rewrite the program so that each statement S writes to a dedicated array A_S. Each dynamic instance of S, $u = (S,\vec{i})$ writes to $A_S(\vec{i})$. Of course, they also need to rewrite all reads so that they obtain values from A_S; this is possible thanks to exact dependence analysis for static control programs [11]. For storage optimization, clearly necessary to avoid drowning in underutilized array elements, they introduce a modulo operation in each dimension. In the optimized code, $(S,\vec{i})$ writes to array element $A_S(\vec{i} \bmod \vec{b})$, where $\vec{b}$ is a vector of positive integers. The product of the elements of b gives the number of memory cells allocated.

For a given statement S, the components of $\vec{b}$ are computed as follows. Identify, for each iteration vector $\vec{i} \in \mathcal{I}_S$, the last operation $L(S,\vec{i})$ that reads the value (if any) computed by the operation $(S,\vec{i})$. Then, compute the (lexicographically) maximal time "delay" between such a write and its last read as $D(S) = \max\{\theta(L(S,\vec{i})) - \theta(S,\vec{i}) \mid \vec{i} \in \mathcal{I}_S\}$. Now, $\vec{i}$ and $\vec{j}$ conflict when $[\theta(S,\vec{i}), \theta(S,\vec{i}) + D(S)]$ and $[\theta(S,\vec{j}), \theta(S,\vec{j}) + D(S)]$ (Lefebvre and Feautrier call these intervals **utility spans**) intersect [1]. When these time intervals (they can be multi-dimensional intervals with the order $\preceq$ if the schedule is multi-dimensional) do not intersect, $(S,\vec{i})$ and $(S,\vec{j})$ don't conflict, and they may store their results in the same memory location.

Now, remember that the operation $(S,\vec{i})$ (resp. $(S,\vec{j})$) is going to write to $A_S(\vec{i} \bmod \vec{b})$ (resp. $A_S(\vec{j} \bmod \vec{b})$). With this in mind, the different moduli are computed successively as follows: the first modulus, b_1, is set to 1 plus the maximal difference $j_1 - i_1$ among all $\vec{i}, \vec{j} \in \mathcal{I}_S$ such that $\vec{i} \bowtie \vec{j}$. With this choice, all operations $(S,\vec{i})$ and $(S,\vec{j})$ such that $\vec{i} \bowtie \vec{j}$ write to different locations since they do not write to the same "row" (i.e., first dimension) of the array A_S unless $j_1 = i_1$. The conflicting iterations for which $j_1 = i_1$ are "disambiguated" by the other dimensions, as follows: b_2 is set to 1 plus the maximal difference $j_2 - i_2$ among all $\vec{i}, \vec{j} \in \mathcal{I}_S$ such that $\vec{i} \bowtie \vec{j}$ and $j_1 = i_1$. The process goes on until all $\vec{i}, \vec{j} \in \mathcal{I}_S$ such that $\vec{i} \bowtie \vec{j}$ have been considered. If the process stops before b_n, the remaining b_i are set to 1 or, equivalently, the corresponding array dimensions are simply removed. Note that, unlike De Greef–Catthoor–De Man, there is no enumeration of the $n!$ permutations of the axes: with the Lefebvre–Feautrier technique, the successive moduli are always computed starting from the outermost loop of the original program.

As mentioned in Section 2.1, this approach subsumes the bounding box approach proposed in [34], which defines b_k as the maximal difference $j_k - i_k$ among all $\vec{i}, \vec{j} \in \mathcal{I}_S$ such that $\vec{i} \bowtie \vec{j}$, i.e., a larger set compared to Lefebvre–Feautrier, which adds the $k-1$ constraints $j_1 = i_1, \ldots, j_{k-1} = i_{k-1}$. We come back to this interesting **successive moduli** mechanism in Section 5. Let us now illustrate it with our running examples.

Example 1 (Cont'd) Note that with the Lefebvre–Feautrier technique, the fact that, in the original program, the array is accessed as $A(i,j)$ is irrelevant. For example, if A was defined as a one-dimensional array, with $A(Ni+j)$ instead of $A(i,j)$, or even if $A(i,j)$ was replaced by $A(f(i,j))$ in both loops, for any injective function f, the memory reduction technique would give the same result.

Consider Figure 2 that depicts the conflicting iterations. The successive moduli approach defines $\vec{b}$ as follows: b_1 is 1 plus the maximal difference for the first dimension between two conflicting iterations, i.e., $b_1 = 2$. Then, it remains to consider all conflicting iterations with same loop counter i and the maximal difference in terms of the loop counter j, and we get $b_2 = 2$. Therefore, $(S,\vec{i})$ writes to the array element $A(i \bmod 2, j \bmod 2)$ with a required memory size equal to 4. This is a bit more than the optimal we found previously (equal to 2) but of the same order of magnitude.

Note that instead of working with the schedule $\theta(S,(i,j)) = Ni+j$, $\theta(T,(i,j)) = Ni+j+1$, we may consider the equivalent multi-dimensional schedule $\theta(S,(i,j)) = (i,j)$ and $\theta(T,(i,j)) = (i,j+1)$

[1] Note that this sometimes unnecessarily over-constrains the problem as we will illustrate with Example 1. It is indeed sufficient to say, as we did previously, that $\vec{i}$ and $\vec{j}$ conflict only when $[\theta(S,\vec{i}), \theta(L(S,\vec{i}))]$ and $[\theta(S,\vec{j}), \theta(L(S,\vec{j}))]$ intersect. Surprisingly, this is also how Lefebvre and Feautrier define the utility spans in the beginning of [21] but not when they present their final algorithm! It seems that this approximation was an implementation choice, maybe motivated by the particular structure of their programs, but they did not justify (or maybe they were not aware of) its consequences on the memory size required by the allocation.

if $0 \leq j < N-1$, $\theta(T,(i,j)) = (i+1,0)$ if $j = N-1$. But this schedule is only piece-wise affine and not affine. And now if, for this schedule, we overconstrain the problem like Lefebvre–Feautrier with the quantity $D(S)$, we would find a much less interesting memory reduction. Indeed, $D(S)$, the maximal "time" difference between a write and its last read, is now equal to $(1, 1-N)$ (see Figure 2 on the right). This does not change b_1, which is still equal to 2. However, now any two indices whose difference is less than $(1, 1-N)$ are considered to conflict, for example $(i,0)$ and $(i, N-1)$, and we get $b_2 = N$ for a memory size equal to $2N$, which is much worse!

One can argue that the schedule θ used here, either one-dimensional with a parameter, or piece-wise affine multi-dimensional, is not an affine multi-dimensional schedule in the sense of Lefebvre–Feautrier, in other words, this code is not considered in their model. This is true, but this means that for such "pipelined" codes, there is, even before the mapping problem, the problem of modeling the notion of "time". □

Example 2 (Cont'd) The reader should now be used to the Lefebvre–Feautrier successive moduli mechanism, so we won't tarry over the computational details. Considering Figure 4, we first get $b_1 = 2$, then $b_2 = N$ with the corresponding mapping $(S, \vec{i}) \mapsto A_S(i \bmod 2, j \bmod N)$, i.e., $(S, \vec{i}) \mapsto A_S(i \bmod 2, j)$, and a memory size equal to $2N$. Again, we almost lose a factor 2 compared to the De Greef–Catthoor–De Man linearization approach.

For the same example after our loop transformation (see the code in Figure 5), Lefebvre–Feautrier avoids the trap into which the De Greef–Catthoor–De Man method falls. Considering Figure 6, we first get $b_1 = N$. Then, since there are no conflicting iterations with the same i, we get $b_2 = 1$, with the corresponding mapping $(S, \vec{i}) \mapsto A_S(i \bmod N, j \bmod 1)$, which leads, after simplifications, to the mapping $(S, \vec{i}) \mapsto A_S(i)$ and a memory size equal to N. We gain an order of magnitude. □

The Lefebvre–Feautrier technique is interesting for several reasons. First, when exact dependence analysis is feasible, they can completely rewrite the code and rely only on the way iterations define new values and not on the original arrays. Second, their successive moduli mechanism exploits the multi-dimensional nature of the problem, it can generate mappings from a larger class than simple canonical linearizations followed by a modulo operation, and it seems very robust, even if the moduli are not computed in a basis aligned with the schedule as illustrated with Example 2 after our loop transformation. We will analyze this mechanism in Section 5 and, in particular, identify when it is indeed efficient. Note however that there are some extreme situations where the Lefebvre–Feautrier approach fails to find a "good" mapping, as the following example shows.

Example 1 (Cont'd) Suppose that the code of Figure 1 is scheduled, not as it is written, but with schedule $i+Nj$ for the first loop and $i+Nj+1$ for the second (i.e., we apply a loop permutation $(i,j) \mapsto (j,i)$ on each loop, and that the second loop is still initiated, as before, one clock cycle after the first loop. The conflicting iterations are the same as those depicted on Figure 2, except that the two axes should be permuted. And now, we get $b_1 = N$, then $b_2 = 2$ for a total memory size $b_1 b_2 = 2N$. In this case, we applied the successive moduli mechanism, considering the axes in the wrong order! □

Again, one can argue that the schedule θ used in the previous example is not really an affine multi-dimensional schedule in the sense of Lefebvre–Feautrier. This is again true, it is affine $(i+Nj)$ but depends on the size parameters (N) in a non-affine way. But, still, such codes exist and we want to be able to handle them and, here, we are analyzing not the Lefebvre–Feautrier model, but its allocation mechanism. One can also argue that, to avoid this problem, we can try all $n!$ permutations of the axes. This is indeed very likely to work. But one can also build extreme cases in which it fails. As we will see in Section 5, for such extreme cases, it is important to be able to choose both the basis and the order in which dimensions are considered before applying the successive moduli mechanism, and this is the weakness of the Lefebvre–Feautrier technique: it relies on one particular basis, and one particular order of dimensions, those given by the original program.

2.3 Quilleré and Rajopadhye

In [26], Quilleré and Rajopadhye studied memory reuse for systems of recurrence equations, a computation model used to represent simple algorithms to be compiled into circuits. In this model, an n-dimensional equation is, at least syntactically, similar to a statement surrounded by n loops in a program: for each index $\vec{i}$ in some "iteration domain", the equation S for index $\vec{i}$ corresponds to a particular value, just like an operation $(S, \vec{i})$ in a program. Quilleré and Rajopadhye propose to store this value in a dedicated array A_S, in location $A_S(f(\vec{i}))$, where f is a linear function from an n-dimensional space (the iteration domain) to a p-dimensional space (the array locations). Clearly, any two values whose index vectors differ by an element of the null space of f are mapped to the same array element. They call such a function a **projection**.

In their study, the size of the iteration domain of an equation is assumed to depend linearly on a parameter (such as N in the previous examples) in all dimensions, and this parameter can be arbitrarily large. What they seek is a linear function f that does not depend on N and such that p is as small as possible, i.e., a valid "projection" onto a linear space of *smallest* dimension. This objective is not exactly the same as ours (reducing the total memory size and not the dimension), but they argue that with their hypotheses reducing the dimension of A_S is the primary optimization criterion for memory reduction. Those hypotheses are, essentially, that iteration domains have large sizes in all dimensions, that the schedule is a multi-dimensional affine schedule, and that conflicting vectors are obtained by approximation with the quantity $D(S)$. Also, a technique difference is that, in their model, they can write to the memory at the same time they read, so to use their technique with our assumptions, we should define $D(S)$ as the maximal delay between a write and the first time where the location can be reused, i.e., the next time after $D(S)$.

They construct valid mapping projections whose rank p is linked to the *depth* d of $D(S)$, i.e., one plus the number of leading zeros in $D(S)$. Their approach is a bit complicated to explain in a short paragraph but, basically, it exploits the fact that the differences $\vec{i} - \vec{j}$ of conflicting indices $\vec{i} \bowtie \vec{j}$ always lie in a subspace of dimension $n - d + 1$ (that is, in general, "thin" in one dimension) since $-D(S) \prec \theta(\vec{i} - \vec{j}) \prec D(S)$. The fact that θ is of full rank allows them to define a memory mapping as a projection onto a subspace of dimension $p = n - d + 1$ (plus a modulo operation in the "thin" dimension). This projection can be obtained by reasoning in a basis defined from the schedule, i.e., so that after this change of basis, the linear part of the schedule is the identity. The next paragraph explains this mechanism more formally.

Suppose, to simplify the explanations, that θ is an n-dimensional schedule (where n is also the dimension of the vectors $\vec{i}$) and that it is given by a unimodular matrix U, i.e., $\theta(S, \vec{i}) = U\vec{i}$. Write $D(S) = \lambda D'(S)$ where $D'(S)$ is a primitive vector (i.e., with coprime components). Since the depth of $D'(S)$ is d, we can complete the set of vectors $\{\vec{e}_1, \ldots, \vec{e}_{d-1}, D'(S)\}$ into a basis of $\mathbb{Z}^n$ ($\vec{e}_i$ is the i-th canonical vector). Let V be the unimodular matrix defined by this basis. Denote by $\vec{m}$ the d-th row of $V^{-1}U$ and by M the $(n-d) \times n$ matrix whose rows are the last $n - d$ rows of $V^{-1}U$. Then, the mapping $\vec{i} \mapsto (\vec{m}.\vec{i} \bmod \lambda, M\vec{i})$ is valid. In other words, if we work in the basis given by the schedule, i.e., with $\vec{i}' = U\vec{i}$, then the mapping is a projection along the first $d-1$ canonical vectors, plus a modulo operation along $D'(S)$. The correctness of this mechanism is easy to understand.

Consider two iterations $\vec{i}$ and $\vec{j}$ mapped to the same location. This means that $\vec{i} - \vec{j}$ belongs to the null space of the mapping, i.e., $U(\vec{i} - \vec{j}) = \sum_{i=1}^{d-1} x_i \vec{e}_i + \lambda x_d D'(S)$. This can be interpreted as a projection along the vectors $U^{-1}\vec{e}_i$, $1 \le i < d$, and what they call a "pseudo-projection" (because of the modulo operation) along $U^{-1}D'(S)$. Now, if $\vec{i}$ and $\vec{j}$ conflict, then the depth of $U(\vec{i}-\vec{j}) = \theta(\vec{i}-\vec{j})$ is at most d, thus $x_i = 0$, for $1 \le i < d$. Therefore $\theta(\vec{i} - \vec{j}) = \lambda x_d D'(S) = x_d D(S)$, and finally $x_d = 0$ since $-D(S) \prec \theta(\vec{i} - \vec{j}) \prec D(S)$, i.e., $\vec{i} = \vec{j}$. This proves that the mapping is valid. [2]

Example 2 (Cont'd) For the code of Figure 3 scheduled with $\theta(i, j) = (i + j, j)$ (see the code of Figure 5), the write corresponding to iteration (i, j) in the original iteration space occurs at iteration $(i+j, j)$ and the read (for iteration $(i+1, j)$ in the original iteration space) occurs at time $(i + j + 1, j)$, thus $D(S) = (1, 0)$. Since we assumed that a write cannot occur at the same time as a read, we need to consider $(1, 1)$ instead of $(1, 0)$ to apply the Quilleré–Rajopadhye technique. Since $\theta^{-1}(i, j) = (i - j, j)$, we find that we can project along $\theta^{-1}(1, 1) = (0, 1)$, and we find the mapping $(i, j) \mapsto i$ as Lefebvre–Feautrier. □

Quilleré and Rajopadhye also mention, as a secondary objective, the use of modulo operations in all dimensions, but only with a brief analysis and more with a bounding box mechanism in mind as Tronçon, Bruynooghe, Janssens, and Catthoor [34] than with a successive moduli mechanism as Lefebvre–Feautrier [21]. So, in our opinion, their main contribution is that they showed the interest to work, not necessarily in the basis given by the original loop indices (like Lefebvre–Feautrier), but in a different basis, and in particular in a basis built from the schedule.

We point out however that to apply this technique, we need that the schedule does express a basis, which is not always the case. For example, the basis that corresponds to the schedule $(i, j) \mapsto (Ni + j)$ we used for Example 1 is "hidden", because this is the linearized version of the schedule $\theta(i, j) = (i, j)$. This is typically the case for codes scheduled with very skewed linearized schedules as those developed in [6]. Also, the technique relies on the hypotheses that the iteration domain is full-dimensional and writes a value at each iteration. If the schedule traverses an iteration domain larger (in size and/or in dimension) than the subdomain where the writes occur then, for extreme cases, the basis given by the schedule is not necessarily the right one in which to project. In other words, to handle more general cases, we need to better understand what is the right basis (if any) in which all these memory reuse mechanisms should be applied.

3 A formal statement of the memory reuse problem

In this section, we build a precise mathematical model of the memory reuse problem for a scheduled program. We define the basic object that we need to build from the scheduled program, a set $\mathcal{C}$ of pairs of **conflicting indices**, which corresponds to data that may not share the same memory locations. Then, we formally introduce linear **modular** allocations, and we prove some basic properties of such mappings.

[2] Quilleré and Rajopadhye count this mapping as d projections but, actually, they forgot that $D'(S)$ may have components proportional to N, especially the first one, so it this not really true that they can always project at least once; there are indeed codes where no memory reuse is possible! So, this is more correct to say that the minimal number of dimensions, for a non parameterized projection, is either $n - d$ or $n - d + 1$ depending whether the first nonzero component of $D'(S)$ is proportional to N or not. For example, on a square of size N, if $D(S) = (0, N)$, then this is as if $D(S)$ was equal to $(1, -N)$, i.e., of depth 1. And if $D(S) = (N, *)$, then there is no real reuse.

3.1 The set of conflicting indices

The first step to model the problem is to decide how we are going to represent the data whose memory allocation need to be optimized. Since we are seeking closed form allocation functions, we cannot describe these data by some kind of enumeration and get mappings with no particular structure, as we would do, for example, by completely unrolling the loops in the code and by allocating data in registers or in a fully-associative cache. We want indeed to develop techniques whose complexity depends on the program structure and not on the number of computations it describes (i.e., a compile-time static optimization), and that can be used directly in the program with accesses to optimized (but standard in their form) arrays. Therefore, we need to represent each data by a symbolic object, typically an index (a vector) $\vec{i}$.

3.1.1 What is an index?

One possibility is to work, as we explained in Section 2.1 for the De Greef–Catthoor–De Man technique, with the original array indices in the program. To optimize an array A, represent each array element $A(\vec{i})$ by its index $\vec{i}$, and seek an allocation function σ_A, which is, mathematically, defined from the set of array indices $\vec{i}$ to memory addresses. In the final program after memory optimization, one just needs to replace each occurrence of $A(\vec{i})$ by an access to the memory address $\sigma_A(\vec{i})$ (plus some base address) or, equivalently, by $L_A(\sigma_A(\vec{i}))$ where L_A is a new one-dimensional array.

Lefebvre and Feautrier (see Section 2.2) as well as Quilleré and Rajopadhye (see Section 2.3) do otherwise. The data whose storage is to be optimized are not the original array elements but the new values created by the program execution (each such value is represented by the statement S that produces it and the iteration vector $\vec{i}$ of the surrounding loops). Define a new array A_S to store the values produced by S and seek an allocation function σ_S, which is defined from the set of iteration vectors $\vec{i}$ to array indices in A_S. The fact that A_S is then stored in row-major order, or any other linearization, has no effect on the storage requirement. In the final program, one needs to replace the left-hand side of S, and each use of the data produced by $(S, \vec{i})$, by $A_S(\sigma_S(\vec{i}))$.

If each statement S in the original program writes to a different array A_S, with an access function f_S (i.e., operation $(S, \vec{i})$ writes to $A_S(f_S(\vec{i}))$, it is *a priori* more general to work with the iteration vectors than with the array indices. Indeed, given a valid mapping σ_A of A_S, the mapping of iterations defined by $\sigma_S(\vec{i}) = \sigma_A(f_S(\vec{i}))$ is also valid. The converse is true only if f is one-to-one. However, working with iteration vectors requires *exact* dependence analysis [11] to be able to rewrite the code, which is not always possible. Therefore, to be as general as possible, and because we are interested in memory reuse mechanisms rather than program analysis and transformation, we will use the generic term **indices** for the vectors (array indices, iteration vectors, or any other representation by vectors) that represent the data whose storage is to be optimized. Only when necessary will we make a distinction between array indices, iteration vectors, or another indexing. Said another way, our memory allocation methods may be applied to any indexing scheme for program data. In particular, when we apply a linear mapping to an index vector, this linearity is *with respect to* the chosen indices; these may be themselves nonlinear functions of, for example, the iteration indices.

3.1.2 Conflicting indices: definition, construction, approximation

Whatever the chosen representation by indices for the data that need to be stored, we need to characterize which indices correspond to data that can be mapped to the same memory location and which indices do not. This is the relation $\bowtie$ of conflicting indices that we introduced for the examples in Section 2.

Definition 1 (Conflicting indices) *Two indices $\vec{i}$ and $\vec{j}$ conflict, denoted by $\vec{i} \bowtie \vec{j}$, if they correspond to two values that are simultaneously live in the program execution given by the schedule θ. By convention, we also say that $\vec{i} \bowtie \vec{i}$.*

We define $CS = \{(\vec{i}, \vec{j}) \mid \vec{i} \bowtie \vec{j}\}$, the **set of all pairs of conflicting indices**. Note that this set is nothing but a multi-dimensional generalization of the interference graph for register allocation. Also, the well-known quantity MAXLIVE (the maximal number of values simultaneously live) is equal to the maximal cardinality of a set S such that $S \times S \subseteq CS$, i.e., a *clique* in graph terminology, all indices in S conflict with each other. To study linear allocations (Section 3.2.2), we also define the **index difference set** $DS = \{\vec{i} - \vec{j} \mid (\vec{i}, \vec{j}) \in CS\}$. Because $\bowtie$ is symmetric, DS is 0-symmetric ($d \in DS$ iff $-d \in DS$). Because we made $\bowtie$ reflexive by definition ($\vec{i} \bowtie \vec{i}$), $0 \in DS$.

Note that, unlike De Greef–Catthoor–De Man, we do not collect the indices that are simultaneously live for a particular time t; we just collect the indices that conflict, whenever this conflict occurs. Lefebvre–Feautrier and Quilleré–Rajopadhye use a notion similar to $\bowtie$, but it is encapsulated in their allocation algorithms. We prefer to highlight the sets CS and DS as mathematical objects, in order to better understand how they affect the quality of the allocation algorithms. Depending on the actual memory allocation algorithm and implementation, one may want to indeed build CS (or DS) first, or to build and use it, on the fly, during the allocation algorithm.

Let us see how we can define the relation $\bowtie$ exactly, for the case where $(S, \vec{i})$ is going to write to $A_S(\vec{i})$. In this case, only data produced by two operations of the same statement S may write to the same location, therefore it is sufficient to represent the data to be stored by the iteration vector $\vec{i}$ of the corresponding operation $(S, \vec{i})$. Denote by $\overline{\mathcal{I}_S}$ the set of iterations for which $(S, \vec{i})$ produces a result to be stored in memory and that will be used later (in practice, $\mathcal{I}_S$ is often equal to $\overline{\mathcal{I}_S}$, but this may not be always the case). For each iteration vector $\vec{i} \in \overline{\mathcal{I}_S}$, as we did in Section 2.2, let $L(S, \vec{i})$ be the last operation that reads the value computed by the operation $(S, \vec{i})$. Then:

$$\vec{i} \bowtie \vec{j} \Leftrightarrow \begin{cases} \vec{i} \in \overline{\mathcal{I}_S}, \, \vec{j} \in \overline{\mathcal{I}_S} \\ \theta(S, \vec{i}) \preceq \theta(L(S, \vec{j})) \\ \theta(S, \vec{j}) \preceq \theta(L(S, \vec{i})) \end{cases} \tag{1}$$

in other words, $\vec{i} \bowtie \vec{j}$ when $\vec{i}$ and $\vec{j}$ correspond to values whose lifetimes overlap; the lifetimes are the intervals $[\theta(S, \vec{i}), \theta(L(S, \vec{i}))]$ and $[\theta(S, \vec{j}), \theta(L(S, \vec{j}))]$.

So far, the formulation of the relation $\bowtie$ by System 1 is purely formal. We did not explain how to compute it. We will not do so here, since this is not the goal of this paper and we want to keep the discussion as general as possible. For completeness, however, we note that when θ is defined by affine schedules (even multi-dimensional), when $\overline{\mathcal{I}_S}$ can be described as all integer points in a polytope, and when all accesses to arrays are affine functions, then $\vec{i} \mapsto L(S, \vec{i})$ is a piece-wise affine function (see [21,26]) and can be obtained with standard techniques to compute lexicographic minima or maxima in polytopes. Finally, CS can be represented as all the integer points in a union of polytopes (when $L(S, \vec{i})$ is piece-wise affine) or simply a polytope (when $L(S, \vec{i})$ is affine). As noted in [26], it may be interesting to first decompose $\overline{\mathcal{I}_S}$ into disjoint subdomains (each operation $(S, \vec{i})$ will then write to a different array, depending on the subdomain $\vec{i}$ belongs to) on which $L(S, \vec{i})$ is affine, so that the corresponding CS for each subdomain can be represented by the integer points in a polytope and not a union of polytopes. Indeed, as we will see in Section 5, heuristics can be guaranteed in the case of polytopes.

The previous technique gives a way to compute the set CS exactly, thanks to an exact analysis of lifetimes. When exact dependence analysis is not possible (or not desired), it is still possible to compute an approximation $\mathcal{C}$ of CS, as long as this approximation is a super-approximation, i.e., $CS \subseteq \mathcal{C}$. For example, when reasoning with array indices, instead of considering the exact lifetimes of the values stored in the array, we can say that an address is live from the first time it is written to the last time it is read (even if in the meantime, it is dead at some point, then written again). With such a definition, there is no need to compute any quantity similar to $L(S, \vec{i})$. Indeed, suppose, to make the discussion simpler, that the program has only two statements S and T, that any operation $(S, \vec{i})$ for $\vec{i} \in \mathcal{I}_S$ writes to $A(f(\vec{i}))$, that any operation $(T, \vec{k})$ for $\vec{k} \in \mathcal{I}_T$ reads $A(g(\vec{k}))$, and that this value is computed by some operation of S (i.e., the array A is not live-in for the program). Then, even when f and g are not one-to-one, we can directly express the approximation $\mathcal{C}$ for CS as follows:

$$\begin{aligned}\mathcal{C} = \{(\vec{a}, \vec{c}) \quad | \quad & \vec{i}, \vec{j} \in \mathcal{I}_S,\, \vec{k}, \vec{l} \in \mathcal{I}_T,\, \vec{a} = f(\vec{i}) = g(\vec{k}),\, \vec{c} = f(\vec{j}) = g(\vec{l}), \\ & \theta(S, \vec{i}) \preceq \theta(T, \vec{l}),\, \theta(S, \vec{j}) \preceq \theta(T, \vec{k})\}\end{aligned}$$

In other words, $A(\vec{a})$ and $A(\vec{c})$ are written, and they are read only later, so they may not share the same location with the chosen definition of liveness.

To conclude this discussion, and to make the rest of the paper simpler, we now forget about the way CS and DS are computed (or super-approximated) and we assume that the set of conflicting indices (resp. index difference set) is represented by a set $\mathcal{C}$ (resp. $\mathcal{D}$) which is exact or a super-approximation. For bounds and heuristics, we consider the "simplest" (though already complex) case: we assume that the index difference set DS is approximated by a set $\mathcal{D}$ and that $\mathcal{D} = \mathring{K}$ for some 0-symmetric polytope K, where $\mathring{K}$ denotes the set of all integer points within K.

Remarks:

- It may happen in practice that, even though $\mathcal{D}$ is not the set of all integer points in a polytope, it is possible to represent it as the intersection of a polytope and a sublattice of $\mathbb{Z}^n$. In this case, we can change the index representation with some one-to-one mapping to come back to a "dense" polytope and the situation we study in the next sections.

- We could generalize this discussion to the general case where several statements (or several arrays) want to share the same memory locations. For that, we can easily generalize the relation $\bowtie$ to conflicting operations (instead of conflicting iteration vectors, i.e. operations of a given statement) or conflicting array elements (instead of array indices). And we could add a dimension to represent different statements (or arrays). However, it is more likely that, in practice, the corresponding sets DS and CS will not be as regular as for the simpler case we discuss, and that approximating them as all integer points in polytopes will lead to mappings that are correct, but sub-optimal. This situation goes out of the scope of this paper and is more related to an inter-array storage optimization than to an intra-array storage optimization (with the terminology of Section 2).

3.2 Memory allocation

Given a scheduling function θ, the memory allocation problem is to map the data computed by the program to a set of memory locations that is as small as possible. In this section, we define the allocation functions we are considering and we characterize valid allocations – those for which the semantics of the program is preserved – with respect to a set $\mathcal{C}$ of indices that correspond to pairs of data that may not share the same location. As explained in the previous section, this set is in general a super-approximation of CS, the exact set of conflicting indices. We conclude with some first general lower bounds for storage requirements.

3.2.1 Allocation functions

We seek a storage allocation σ that specifies for each data, represented by an n-dimensional vector $\vec{i}$ (what we called its index), where it will be stored. The main objective is to minimize storage requirements by reusing memory when a value that has been stored is no longer used, following the model assumptions of Section 3.1. The next definition defines a storage allocation as a function σ that maps iteration indices onto array elements of a dedicated array of dimension p.

Definition 2 *An* **allocation** *(or* **mapping***)* σ *of* **size** *size*$(\sigma) = m$ *(i.e., requiring* m *memory locations) is a function* $\sigma : \mathbb{Z}^n \to \mathcal{M}$ *where* $\mathcal{M} \subset \mathbb{Z}^p$ *is a finite set of* m *elements. We say that two allocations* σ *and* τ *are* **equivalent** *if and only if there exists a bijection* ϕ *such that* $\tau = \phi \circ \sigma$.

Two equivalent allocations require the same memory size and they are identical up to a renaming of array elements. As we recall in Section 3.3, this definition follows the theory of **memory skewing schemes** (see [38, Def. 4.1]). To simplify the search space as well as the resulting code generation, it is convenient to restrict the study to linear allocations.

Definition 3 *A* **linear allocation** *(or* **mapping***)* σ *of* **size** *size*$(\sigma) = m$ *is a homomorphism* $\sigma : \mathbb{Z}^n \to \mathcal{M}$ *where* $\mathcal{M} \subset \mathbb{Z}^p$ *is a finite* $\mathbb{Z}$*-module (i.e., finite Abelian group) of* m *elements.*

Two linear and equivalent allocations σ and τ with $\tau = \phi \circ \sigma$ have the same kernel (the kernel of a homomorphism σ is the set $\ker(\sigma) = \{\vec{i} \mid \sigma(\vec{i}) = 0\}$ where 0 is the identity element of $\mathcal{M}$). Conversely, two linear allocations that have the same kernel are equivalent. Indeed, if σ is a homomorphism from $\mathbb{Z}^n$ to some $\mathbb{Z}$-module, then there is an isomorphism $\overline{\sigma} : \mathbb{Z}^n/\ker(\sigma) \to \mathrm{im}(\sigma)$ such that $\sigma = \overline{\sigma} \circ i$ where i is the canonical injection from $\mathbb{Z}^n$ to $\mathbb{Z}^n/\ker(\sigma)$. If σ and τ are two homomorphisms with the same kernel, then $\tau = \overline{\tau} \circ i = \overline{\tau} \circ \overline{\sigma}^{-1} \circ \overline{\sigma} \circ i = \phi \circ \sigma$ with $\phi = \overline{\tau} \circ \overline{\sigma}^{-1}$, thus σ and τ are equivalent. Therefore, two linear allocations are equivalent if and only if they have the same kernel.

The allocation techniques we explained in Section 2 are all using restricted forms of modular mappings. A **modular mapping** $(M, \vec{b})$ stores the value represented by $\vec{i}$ in a p-dimensional array in the array element with indices $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$, where M is a $p \times n$ integer matrix and $\vec{b}$ is a p-dimensional integer vector with nonzero entries (the modulo operation is applied componentwise). The array is implemented as an array with multi-dimensional extent $\vec{b}$, and its size m (that we want to minimize) is the product $\prod_{i=1}^{p} b_i$. This corresponds to $\mathcal{M} = \mathbb{Z}/b_1\mathbb{Z} \oplus \ldots \oplus \mathbb{Z}/b_p\mathbb{Z}$ in Definition 3. A projection corresponds to the case where one or more components of the modulus vector $\vec{b}$ are equal to 1 (in which case, we can forget about the corresponding row(s) of M, see Section 4.1). Note that a modular allocation is a linear allocation and, as we show in the next section, any linear allocation is equivalent to a modular allocation.

3.2.2 $\mathcal{C}$-valid allocations

We now characterize valid allocations, i.e., allocations for which the semantics of the program are preserved. As explained in Section 3.1, we assume that the semantic restriction on the allocation is represented by a set $\mathcal{C} \subset \mathbb{Z}^n \times \mathbb{Z}^n$ of index pairs such that $CS \subseteq \mathcal{C}$. Validity is defined with respect to $\mathcal{C}$ and implies the validity with respect to CS, which in turn means that the allocation does not map conflicting indices to the same memory location.

Definition 4 *An allocation σ is valid for $\mathcal{C}$ (or $\mathcal{C}$-valid) iff $(\vec{i}, \vec{j}) \in \mathcal{C}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$.*

The following is immediate from Definition 2.

Proposition 1 *If σ and τ are two equivalent allocations then σ is $\mathcal{C}$-valid iff τ is $\mathcal{C}$-valid.*

For linear allocations, we can give a validity criterion using $\mathcal{D}$, the set of all differences of conflicting indices, $\mathcal{D} = \{\vec{i} - \vec{j} \mid (\vec{i}, \vec{j}) \in \mathcal{C}\}$. From the definition of CS, $0 \in \mathcal{D}$ and $\mathcal{D}$ is 0-symmetric (symmetric with respect to 0). A linear allocation σ is valid iff $(\vec{i}, \vec{j}) \in \mathcal{C}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i} - \vec{j}) \neq 0$, or equivalently $\vec{d} = \vec{i} - \vec{j}$, $(\vec{i}, \vec{j}) \in \mathcal{C}$, $\sigma(\vec{d}) = 0 \Rightarrow \vec{d} = \vec{0}$. Thus, a linear allocation is $\mathcal{C}$-valid iff $\vec{0}$ is the only integer vector in $\mathcal{D}$ whose image is 0.

Proposition 2 *A linear allocation is $\mathcal{C}$-valid iff $\mathcal{D} \cap \ker \sigma = \{\vec{0}\}$.*

From these definitions, it is easy to derive some lower bounds for the smallest storage size achievable by a $\mathcal{C}$-valid allocation and by a $\mathcal{C}$-valid modular allocation. We have:

$$\min\{\text{size}(\sigma) \mid \sigma \text{ valid for } \mathcal{C}\} \geq \max\{\text{Card}(S) \mid S \times S \subseteq CS\}$$

(Card(S) is the number of integer points in S) since all points in such a S must be mapped to different locations. Note that, when the program is in dynamic single assignment form (each indexed element is written once), $\max\{\text{Card}(S) \mid S \times S \subseteq CS\}$ is the definition of MAXLIVE and there exists an allocation that reaches this lower bound, the greedy assignment obtained when executing the program. But, of course, the corresponding addressing functions will be much more complicated, and not necessarily given by a closed-form formula. [3]

Moreover, if σ is a *linear* $\mathcal{C}$-valid allocation and there is a set S such that $S - S \subseteq \mathcal{D}$, then size$(\sigma) \geq$ Card(S). Indeed, let $\vec{i} \in S$ and $\vec{j} \in S$ such that $\sigma(\vec{i}) = \sigma(\vec{j})$. We get $\sigma(\vec{i} - \vec{j}) = 0$ (because σ is linear), then $\vec{i} = \vec{j}$ for $\ker(\sigma) \cap \mathcal{D} = \{\vec{0}\}$, in other words, σ is $(S \times S)$-valid.

We come back to our examples to illustrate Proposition 2 and these lower bounds, i.e., the link between valid linear allocations and the set $\mathcal{D}$.

Example 1 (Cont'd) For the code of Figure 1, there are only five different vectors in the set DS, $(0, 0)$, the vectors $(0, 1)$ and $(1, 1 - N)$, and their negations. This set is depicted in Figure 7 as well as a polytope K such that $DS = \mathring{K}$. In grey, some regularly spaced points are represented, these are the elements of the kernel of the mapping $(i, j) \mapsto (i \bmod 2, j \bmod 2)$, whose intersection with DS is equal to $\{\vec{0}\}$ and which is thus a valid mapping.

In Figure 8, we represent the set $\mathcal{D}$ that is obtained if we use the approximation based on the maximal time difference $D(S)$ between a write and its read, for a statement S, as explained in Section 2.2. Here, the maximal time difference is $(1, 1 - N)$ and the set $\mathcal{D}$ is the set of all possible differences of iteration vectors in the square of size N that are less than $(1, 1 - N)$. With this super-approximation, the mapping $(i, j) \mapsto (i \bmod 2, j \bmod 2)$ is no longer valid. Actually, since the set $S = \{(0, 0), (0, 1), \ldots (0, N - 1)\}$ is such that $S - S \subseteq \mathcal{D}$, any valid linear allocation requires at least N memory cells. Therefore, with this apparently slight approximation, we already see that we lose an order of magnitude (when N is large) in memory size compared to the optimal solution (which, as seen before, is 2). $\square$

[3] Also, with reasonable hypotheses, one can get a mapping reaching MAXLIVE and given by a closed-form formula using Ehrhart polynomials, combined with some hardware mechanisms (similar to [1]). But this goes out of the scope of this paper.

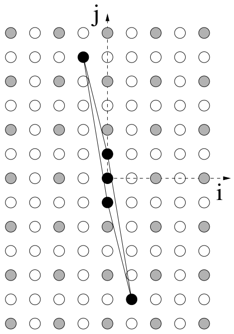

Figure 7: The exact DS for Example 1.

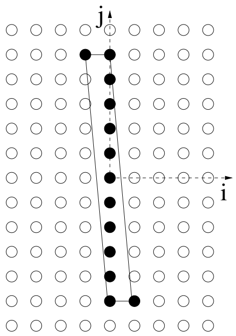

Figure 8: An approximation $\mathcal{D} = \mathring{K}$ of DS.

Example 2 (Cont'd) For the code of Figure 3, the set DS is the set depicted in Figure 9 (compare with Figure 4 to understand how we built it). In grey, we depicted the points that belong to the kernel of the mapping $(i, j) \mapsto (i \bmod 2, j \bmod N)$ that Lefebvre and Feautrier would find (see Section 2.2). The set $S = \{(0, 0), \dots, (0, N-1), (-1, N-1)\}$, of cardinality $N+1$, is such that $S - S \subseteq DS$ and thus shows that any linear allocation requires at least $N+1$ memory cells.

Figure 10 shows the set DS when the schedule $\theta(i, j) = (i + j, j)$ is used, as well as the points that belong to the kernel of the mapping $(i, j) \mapsto i \bmod N$ that Lefebvre and Feautrier find. This mapping is an optimal linear mapping since the set $S = \{(0, 0), (-1, 1), \dots, (1-N, N-1)\}$, of cardinality N, is such that $S - S \subseteq DS$. □

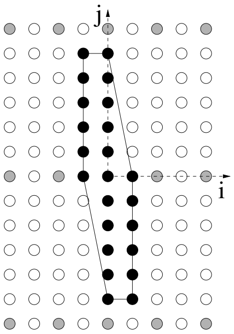

Figure 9: Exact DS for Example 2.

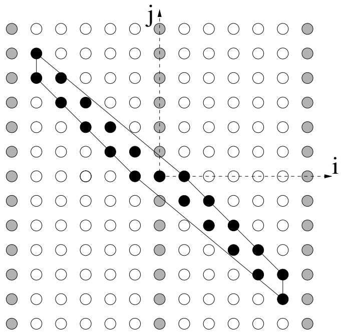

Figure 10: Same with schedule $\theta(i, j) = (i + j, j)$.

3.3 The theory of parallel memories and templates

We should make it clear that linear allocations are not new. They were introduced in 1971 by Budnik and Kuck [2], and called memory **skewing schemes**, for defining data layouts that allow parallel accesses to memory following particular patterns (called *templates*). This concept was studied in more detail by Shapiro [28], then by Wijshoff and van Leeuwen (see in particular [38] and many other research reports and papers by the same authors), and several other authors. The constraints for validity of such allocations are different than for memory reuse; this perhaps explains why this work on linear allocation has been forgotten (as in our previous paper [7]). However, we think it is important to briefly present the similarities and differences between memory allocation for memory reuse in scheduled programs (our problem) and memory allocation for templates.

A skewing scheme s is a mapping from $\mathbb{Z}^n$ to $[0, m-1]$, which maps indices of array elements to

m memory locations. These schemes have been studied in relation with templates (data-templates).

Definition 5 *A template T is any finite subset $T = \{\vec{0}, \vec{t}_1, \ldots, \vec{t}_l\}$ of $\mathbb{Z}^n$ containing $\vec{0}$. An instance $T(\vec{x})$ of T, $\vec{x} \in \mathbb{Z}^n$, is $T(\vec{x}) = \{\vec{x}, \vec{x} + \vec{t}_1, \ldots, \vec{x} + \vec{t}_l\}$.*

We have defined valid allocations with respect to a set $\mathcal{C}$ of index pairs. The validity of skewing schemes, initially related to parallel memory accesses, is defined with respect to a template T and a set Λ (in general, $\Lambda = \mathbb{Z}^n$). A scheme s is (T, Λ)-valid if and only if for any $\vec{x} \in \Lambda$, the restriction of s to $T(\vec{x})$ is an injection. Note that a template corresponds to the notion of clique in an interference graph, i.e., all elements in a template must be mapped to a different location. In other words, the corresponding relation $\bowtie$ is *transitive* within a template, which is not necessarily the case with a description of conflicting indices with $\mathcal{C}$.

For a *linear* skewing scheme, one can also define a set $DT = \{\vec{i} - \vec{j} \mid \vec{i}, \vec{j} \in T\}$ of conflicting differences. Then, a skewing scheme s is $(T, \mathbb{Z}^n)$-valid iff $\ker(s) \cap DT = \{\vec{0}\}$ as in Proposition 2. Therefore, there is indeed a direct correspondence between linear allocation for memory reuse and linear skewing schemes. The techniques we propose in this paper can be used to derive linear skewing schemes valid for templates expressed as or approximated by polytopes. Conversely, we could hope to reuse some results of the theory of linear skewing schemes when $\mathcal{D}$ is equal to the difference set of some template T, i.e., $\mathcal{D} = T - T$. Unfortunately, the main results in the literature are for templates of *small* sizes, and the theory focuses on the mathematical properties of such allocations (properties that we will also use in Section 4) but not at all on the *algorithmic* problem of how to effectively build an allocation that requires the smallest memory size. Actually, since the template is small and described by enumeration, it was considered to be easy to find the optimal allocation by enumeration (see [37] and Section 4 hereafter). But this is not always satisfying in our situation when the size of the difference set can be large, when this set may not specified by enumeration but by some representation whose size does not depend on the number of points it contains (for example a polytope), and when this set can even be parameterized. However, one spectacular (but apparently useless for us) result developed at this time [36] is that, in $2D$, when the template T is a polyomino (i.e., a rook-wise connected set of integer points) of size m, an optimal valid skewing scheme requires exactly m memory cells iff T tessellates the plane and iff there is valid *linear* skewing scheme requiring m memory cells. Also in [31], lower and upper bounds for the optimal allocation are studied, for several complex shapes of templates, but only in dimension 1, i.e., in $\mathbb{Z}$, while we mainly work in higher dimensions.

Finally, we also point out that the theory of skewing schemes also considers nonlinear allocations (in particular *diamond schemes* [16] and *multi-periodic schemes* [32]) and a more general notion of validity, extended to a collection $\mathcal{T} = \{T^{(1)}, \ldots, T^{(r)}\}$ of templates. A scheme s is $(\mathcal{T}, \Lambda)$-valid if and only if, for all $1 \leq k \leq r$, the scheme s is $(T^{(k)}, \Lambda)$-valid. With these notions, we could also hope to reuse some results to derive nonlinear allocations (even though again there is no algorithmic way of deriving such valid allocations in the literature). Indeed, if we consider $\mathcal{C} = \bigcup_{1 \leq k \leq r} \bigcup_{\vec{x} \in \Lambda} \left(T^{(k)}(\vec{x}) \times T^{(k)}(\vec{x})\right)$ (allowing here unbounded iteration domains), we see that a $\mathcal{C}$-valid allocation σ is a $(\mathcal{T}, \Lambda)$-valid scheme. Conversely, considering the template collection $\mathcal{T} = \{\{\vec{i}, \vec{j}\} \mid (\vec{i}, \vec{j}) \in \mathcal{C}\}$ and $\Lambda = \{\vec{0}\}$, a $(T, \{\vec{0}\})$-valid scheme is a $\mathcal{C}$-valid allocation. The latter shows that, though allocations for memory reuse and skewing schemes for templates may share concepts, the constraints that they respectively need to respect are somehow different. In the above correspondence, only unions of sets of the form $S \times S$ (i.e., cliques in terms of interference graphs) are used for building valid skewing schemes and valid allocations for memory reuse only use trivial template instances. Valid skewing schemes may be useful only for constructing $\mathcal{C}$-valid allocations with $\mathcal{C}$ having some regularity.

Example 1 (Cont'd) We have $CS \subseteq \mathcal{C} = \bigcup_{1\le k\le 2} \bigcup_{\vec{x}\in\mathbb{Z}^2} \left(T^{(k)}(\vec{x}) \times T^{(k)}(\vec{x})\right)$ with templates $T^{(1)} = \{\{(0,0),(0,1)\}\}$ and $T^{(2)} = \{\{(0,N-1),(1,0)\}\}$. All the vectors $\vec{x} \in \Lambda = \mathbb{Z}^2$ are used for generating $\mathcal{C}$ from $T^{(1)}$ and $T^{(2)}$. For linear allocations, the validity only needs to be checked with respect to $T^{(1)}$ and $T^{(2)}$. For instance, the restrictions of the valid mapping $(i,j) \mapsto Ni + j \bmod 2$ to $T^{(1)}$ and $T^{(2)}$ are injective. □

To conclude this section, there is indeed a connection between allocation for memory reuse and skewing schemes for templates, but the constraints on validity, their size and representation, and the complexity issues are quite different.

4 Integer lattices and linear allocations

Now and for the rest of the paper, we restrict ourselves to the study of linear allocations. We rely on a lattice-based approach that leads to a unified framework for studying previously proposed allocation mechanisms and introducing new ones. Let $\vec{a}_1, \dots, \vec{a}_n$ be n linearly independent points in $\mathbb{R}^m$. The set Λ of points $\vec{x} = u_1\vec{a}_1 + \dots + u_n\vec{a}_n$ where $u_1, \dots, u_n$ are integers is called **lattice** of rank n. The system of points $(\vec{a}_1, \dots, \vec{a}_n)$ is a basis of Λ; for $n = m$, $\det(\vec{a}_1, \dots, \vec{a}_n)$, a quantity that does not depend on the choice of a basis for Λ, is called the determinant of Λ, denoted by $d(\Lambda)$. We write $\vec{x} = A\vec{u}$ where A is the matrix with column vectors $\vec{a}_1, \dots, \vec{a}_n$ and $\Lambda = A\mathbb{Z}^n$. Lattices are useful since, as we will see, they are in correspondence with linear allocations. The $\mathcal{C}$-valid linear allocations further correspond to a special class of lattices that we call **strictly admissible lattices** for $\mathcal{D}$.

4.1 Kernels and representation of linear allocations

Proposition 2 shows that a linear allocation σ is $\mathcal{C}$-valid iff its kernel has no point in $\mathcal{D}$ other than $\vec{0}$. The kernel of a linear allocation σ from $\mathbb{Z}^n$ to $\mathcal{M}$ is a sublattice Λ of $\mathbb{Z}^n$ called the **underlying lattice** of σ. The rank of Λ is n, otherwise $\mathbb{Z}^n/\Lambda$ would be infinite, which would contradict the fact than $\mathcal{M}$ is finite. Given a modular mapping $(M, \vec{b})$, a basis for Λ can be built by integer matrix computations (see for example [5]). We also noticed that two linear allocations are equivalent iff they have the same kernel, hence two linear allocations are equivalent iff they have the same underlying lattice. Thus equivalent linear allocations may be studied through their underlying lattice. Since equivalent allocations are equal except for a bijection between their images, we may also reduce their study to the study of a unique "good" (with respect to the storage requirements) class representative.

We first need a few classical results on integer matrices [25]. Given a matrix M of rank n in $\mathbb{Z}^{n\times n}$, there exist a unimodular (i.e., with determinant 1 or -1, thus with an integer inverse) matrix U and an upper triangular matrix $H \in \mathbb{Z}^{n\times n}$ such that $H = MU$; moreover, H can be chosen such that all entries $h_{i,j}$, $j > i$, are in a fixed residue system modulo $h_{i,i}$, $1 \le i \le n$. The matrix H, called the **Hermite (normal) form** of M, is uniquely determined and is not changed if M is multiplied on the right by a unimodular matrix. Thus, H is determined by the lattice $M\mathbb{Z}^n$ and not its basis. (We can also choose H to be lower triangular instead of upper triangular, with similar properties.) Given $M \in \mathbb{Z}^{n\times n}$ of rank n, there exist two unimodular matrices U_1 and U_2 such that $S = U_1 M U_2$ is diagonal in $\mathbb{N}^{n\times n}$, $S = \mathrm{diag}(s_1, s_2, \dots, s_n)$, with $s_1|s_2$, …, $s_{n-1}|s_n$; S is uniquely determined, is not changed if M is multiplied on either side by a unimodular matrix, and is called the **Smith (normal) form** of M.

Proposition 3 *Let $\Lambda \subseteq \mathbb{Z}^n$ be a lattice of rank n. There exists a modular mapping $(U, \vec{s})$ whose kernel is Λ, with U unimodular in $\mathbb{Z}^{n\times n}$, and $\vec{s}$ is such that $d(\Lambda) = \prod_{i=1}^{n} s_i$. Furthermore, any linear allocation σ is equivalent to a modular mapping $(U, \vec{s})$.*

Proof. For the first statement of the proposition, consider a matrix $A \in \mathbb{Z}^{n\times n}$ whose columns form a basis of Λ. Write $S = U_1 A U_2$, the Smith form of A. Then one can take $U = U_1$ and $\vec{s}$ the vector such that $S = \mathrm{diag}(\vec{s})$. Indeed $U\vec{x} \bmod \vec{s} = 0$ iff there exists $\vec{u} \in \mathbb{Z}^n$ such that $U\vec{x} = S\vec{u}$, iff there exists $\vec{v} \in \mathbb{Z}^n$ (with $\vec{v} = U_2\vec{u}$) such that $\vec{x} = U_1^{-1} S U_2^{-1} \vec{v} = A\vec{v}$. In other words, $(U, \vec{s})$ is a modular mapping whose kernel is Λ. Since A and S are unimodular equivalent then $d(\Lambda) = \det S = \prod_{i=1}^{n} s_i$. The second statement of the proposition is proved by considering $\Lambda = \ker(\sigma)$; $(U, \vec{s})$ has kernel Λ and is thus equivalent to σ. ⊠

A modular mapping $(M, \vec{b})$ stores the value corresponding to $\vec{i}$ in a multi-dimensional array, at the location indexed by $M\vec{i} \bmod \vec{b}$ (the modulo operation is applied componentwise). The rows of M corresponding to the unit entries in $\vec{b}$ may be omitted since the corresponding dimension in the storage array is not used. We now show that Proposition 3 leads to a modular mapping that uses the smallest number of dimensions among all modular mappings with the same kernel.

Lemma 1 *Let $S = diag(1, \dots, 1, s_1, \dots, s_p)$ of rank n in $\mathbb{Z}^{n\times n}$ be in Smith form with exactly p non-unit diagonal entries. Any diagonal matrix $B = PSQ$, with P and Q nonsingular matrices in $\mathbb{Z}^{n\times n}$, has more than p non-unit entries in its diagonal.*

Proof. For a matrix A and sets I, J of indices with $|I| = |J|$, we denote by $A_{I,J}$ the determinant of the submatrix of A built on rows I and columns J. Assume that B has strictly less than p non-unit entries in its diagonal. This means that there exists a set I of $n-p+1$ distinct indices such that the corresponding $(n-p+1)\times(n-p+1)$ minor $B_{I,I}$ of B is unity. By the Binet-Cauchy formula (see [25]), we have

$$B_{I,I} = 1 = \sum_{J,K} P_{I,J} S_{J,K} Q_{K,I} = \sum_{J} P_{I,J} S_{J,J} Q_{J,I},$$

where the first sum is taken over all sets J and K of $n-p+1$ indices, and the second sum over all sets J of $n-p+1$ (indeed, $S_{J,K} = 0$ except if $J = K$). Any $(n-p+1)\times(n-p+1)$ minor of S is an integer multiple of s_1, hence the same is true for all the terms of the latter sum, which contradicts the fact that $B_{I,I}$ is equal to one. ⊠

To the linear allocation σ, we now associate a modular mapping $(U_p, \vec{s}_p)$, where $U_p \in \mathbb{Z}^{p\times n}$ is formed by the last p rows of U as in Proposition 3, and where $\vec{s}_p$ is given by the non-unit entries of $\vec{s}$.

Theorem 1 *Among all modular mappings equivalent to σ, $(U_p, \vec{s}_p)$ requires the smallest number of array dimensions and the least storage, namely $d(\Lambda) = s_1 s_2 \dots s_p$. Furthermore, any other modular mapping $(M, \vec{b})$ equivalent to σ is such that $\prod_i b_i$ is a multiple of $d(\Lambda)$.*

Proof. Let $(U, \vec{s})$ be the mapping of Proposition 3. By construction, $U^{-1}S$, with $S = \mathrm{diag}(\vec{s})$, is a basis of the kernel Λ of σ. Let $(M, \vec{b})$ be a modular mapping equivalent to σ. We show that $B = \mathrm{diag}(\vec{b})$ and $S = \mathrm{diag}(\vec{s})$ are as in Lemma 1.

Write $S_{\vec{b}} = \mathrm{diag}(\vec{b}_S) = U_1 B U_2$, the Smith form of B. Then $(N, \vec{b}_S)$, with $N = U_1 M$, is also equivalent to σ. Indeed, $\vec{x} \in \ker((N, \vec{b}_S))$ iff $N\vec{x} \bmod \vec{b}_S = 0$ iff there exists $\vec{u} \in \mathbb{Z}^n$ such that $N\vec{x} = S_{\vec{b}}\vec{u}$ iff there exists $\vec{u} \in \mathbb{Z}^n$ such that $M\vec{x} = BU_2\vec{u}$ iff there exists $\vec{v} \in \mathbb{Z}^n$ (with $\vec{v} = U_2\vec{u}$) such that $M\vec{x} = B\vec{u}$, thus iff $\vec{x} \in \ker((M, \vec{b}))$. Now, let $H = NV$, with V unimodular, be the

Hermite form of N. Since H is upper triangular and $S_{\vec{b}}$ is in Smith form with diagonal entries satisfying the divisibility property, each column of $NVS_{\vec{b}}$ is equal to 0 modulo $\vec{b}_S$, i.e., each column of $VS_{\vec{b}}$ belongs to the kernel of σ. Hence, there exists $R \in \mathbb{Z}^{n\times n}$ such that $VS_{\vec{b}} = (U^{-1}S)R$, i.e., $S_{\vec{b}} = V^{-1}U^{-1}SR$. Using $S_{\vec{b}} = U_1BU_2$, we obtain as announced that $B = PSQ$ for P and Q in $\mathbb{Z}^{n\times n}$. From Lemma 1, we conclude that $\vec{b}$ must have more unit entries than $\vec{s}$, and that $\prod_i b_i$ is an integer multiple of $\det(S) = d(\Lambda) = s_1 s_2 \dots s_p$. ⊠

The study of linear allocations $\sigma : \mathbb{Z}^n \to \mathcal{M}$ thus reduces to the study of modular mappings $\vec{i} \mapsto U\vec{i} \bmod \vec{s}$ where $U \in \mathbb{Z}^{p\times n}$ can be completed to a unimodular matrix and where $\vec{s}$ has no unit entries. To compute a minimal mapping equivalent to a given linear mapping σ, we just have to compute its kernel, for example as in [5], then a minimal mapping representative as in Theorem 1. Note that this minimal mapping representative is not unique. For example, in the construction of U, one may consider any left unimodular multiplier for the Smith form of A, a basis of the kernel. In addition, the modulus vector need not be in Smith form as long as it has no more than p non-unit entries.

4.2 Strictly admissible integer lattices

Let K be an arbitrary set in $\mathbb{R}^n$. A lattice is called **admissible** for K if it has no point $\neq \vec{0}$ in the interior of K. It is called **strictly admissible** for K if it does not contain a point $\neq \vec{0}$ in K. The quantity $\Delta(K) = \inf\{d(\Lambda) \mid \Lambda \text{ strictly admissible for } K\}$ is called the **critical determinant** of K. We also define $\Delta^0(K) = \inf\{d(\Lambda) \mid \Lambda \text{ admissible for } K\}$ and the quantity we are really interested in, which is $\Delta_{\mathbb{Z}}(K) = \inf\{d(\Lambda) \mid \Lambda \subseteq \mathbb{Z}^n \text{ strictly admissible for } K\}$. While $\Delta(K)$ may not be attained, $\Delta_{\mathbb{Z}}(K)$ is an integer and there always exists a strictly admissible integer lattice Λ such that $d(\Lambda) = \Delta_{\mathbb{Z}}(K)$. The next proposition gives the correspondence between valid linear allocations and strictly admissible lattices.

Proposition 4 *A linear allocation is $\mathcal{C}$-valid iff its underlying lattice is strictly admissible for $\mathcal{D}$. A strictly admissible lattice Λ for $\mathcal{D}$ is the underlying lattice of a $\mathcal{C}$-valid modular mapping that uses $d(\Lambda)$ memory allocations.*

Proof. From Proposition 2, a linear allocation σ is valid iff $\mathcal{D} \cap \ker(\sigma) = \mathcal{D} \cap \Lambda = \{\vec{0}\}$. Given a strictly admissible lattice Λ, Proposition 3 constructs a mapping $(U, \vec{s})$ of storage size $d(\Lambda)$, which is $\mathcal{C}$-valid since its kernel is Λ. ⊠

This gives dual views of the allocation problem. In Sections 5.2 and 5.3, we build a strictly admissible integer lattice, reasoning in the space where $\mathcal{D}$ is defined, and deduce a valid mapping. In Section 5.4, we build the mapping directly, reasoning with a matrix M and a vector $\vec{b}$.

In the remainder of the paper, we focus on the crux of the problem, the quantity $\Delta_{\mathbb{Z}}(K)$ for a 0-symmetric convex body K with positive volume. This corresponds to the case where the set of conflicting differences DS is represented or super-approximated by a set $\mathcal{D} = \mathring{K}$ of integer points within a 0-symmetric convex polytope K. We consider that K can be any such polytope, i.e., we ignore the fact that, for our original problem, K is built in some particular way (from affine schedules for example) and from particular programs. In practice, some particular cases can of course be easier to solve, however, note that, at least in theory, any polytope K can indeed represent the set of conflicting differences of a particular program: simply consider the case of n nested loops, storing (with no possible reuse) a portion – equal to a translated copy of $K/2$ – of an n-dimensional array; in this case, all indices are conflicting and the corresponding set of conflicting differences is all the integer points in $K/2 - K/2 = K$ (this relation because K is 0-symmetric). A typical example would be a loop that accesses only the three main diagonals of a 2D square array of size N, and we should be able to find that, among others, the mapping $(i, j) \mapsto (i, j \bmod 3)$ is enough to store $3N$ values instead of N^2.

What we seek here is a general framework that defines the problem and general mechanisms that are robust enough to handle well even extreme cases that we may encounter, for example cases with "skewed" schedules as used in [6]. We do, however, make two important assumptions. First, to simplify implementation, we assume that K is a rational polytope, i.e., defined by integer linear inequalities. Second, to be able to evaluate the performance of heuristics, we assume that K is full-dimensional and *contains n linearly independent integer points*, hence $\mathrm{Vol}(K) \geq 1$. Otherwise, we can make a preliminary change of basis (at least conceptually for the moment) and work with the polytope K', the intersection of K with the smallest vector subspace that contains all integer points in K. This will allow us to talk about the volume of K' and to get upper bounds in terms of this volume. Without this admittedly technical assumption, such bounds are impossible, since $\Delta_{\mathbb{Z}}(K)$ is a positive integer while the n-dimensional volume of K, even though positive, can be arbitrarily small.

4.3 Exploring the quantity $\Delta_{\mathbb{Z}}(K)$

Many theoretical results exist for the quantities $\Delta(K)$ and $\Delta^0(K)$ (see [13] for an extensive study), but it is an open problem to be able to compute $\Delta(K)$ for most bodies K, even simple ones. Possibly, however, the combinatorial nature of $\Delta_{\mathbb{Z}}(K)$ can make the problem easier. But, as Proposition 5 shows, for large bodies, computing $\Delta(K)$ and $\Delta_{\mathbb{Z}}(K)$ are equivalent problems. Our goal will be then to try to adapt to $\Delta_{\mathbb{Z}}(K)$ well-known lower and upper bounds for $\Delta(K)$, and to discuss heuristics within this framework.

We have $\Delta^0(K) \leq \Delta(K) \leq \Delta_{\mathbb{Z}}(K)$. Theorem 17.3 in [13] states that, if K is a star body [4], $\Delta(K) = \Delta^0(K)$. Proposition 5 below shows that for large star bodies, $\Delta_{\mathbb{Z}}(K) \approx \Delta(K)$. Before we tackle it, note the following important scaling properties: for all $\lambda > 0$, $\Delta(\lambda K) = \lambda^n \Delta(K)$ while $\Delta_{\mathbb{Z}}(\lambda K) \leq \lceil \lambda \rceil^n \Delta_{\mathbb{Z}}(K)$.

Proposition 5 *For a bounded star body K,* $\lim_{\lambda \to \infty} \dfrac{\Delta_{\mathbb{Z}}(\lambda K)}{\Delta(\lambda K)} = 1$

Proof. Let $\epsilon > 0$ and let Λ be a strictly admissible lattice for K with $d(\Lambda) \leq \Delta(K)(1 + \epsilon)$. Define the quantity $\lambda_1(K, \Lambda) = \inf\{\lambda > 0 \mid \lambda K \cap \Lambda \neq \{\vec{0}\}\}$. We have $\lambda_1(K, \Lambda) > 1$. Since, for a bounded star body, $\lambda_1(K, \Lambda)$ depends continuously on Λ (see [13, p. 185]), there exists a rational matrix A_ϵ with $\lambda_1(K, A_\epsilon \mathbb{Z}^n) > 1$ (thus strictly admissible for K) and $\det(A_\epsilon) \leq \Delta(K)(1 + 2\epsilon)$. Let μ_ϵ be the common denominator of the coefficients of A_ϵ: $\mu_\epsilon A_\epsilon \mathbb{Z}^n$ is a sublattice of $\mathbb{Z}^n$, strictly admissible for $\mu_\epsilon K$, and therefore strictly admissible for any μK with $\mu \leq \mu_\epsilon$. Now, let $\lambda > 0$ and let $k = \lceil \frac{\lambda}{\mu_\epsilon} \rceil$, i.e., $(k-1)\mu_\epsilon < \lambda \leq k\mu_\epsilon$. The lattice $k\mu_\epsilon A_\epsilon \mathbb{Z}^n$ is integer and strictly admissible for λK. Furthermore:

$$\Delta(\lambda K) \leq \Delta_{\mathbb{Z}}(\lambda K) \leq \det(k\mu_\epsilon A_\epsilon) \leq k^n \mu_\epsilon^n \Delta(K)(1 + 2\epsilon) \leq (\lambda + \mu_\epsilon)^n \Delta(K)(1 + 2\epsilon)$$

Therefore:

$$1 \leq \frac{\Delta_{\mathbb{Z}}(\lambda K)}{\Delta(\lambda K)} \leq (1 + \frac{\mu_\epsilon}{\lambda})^n (1 + 2\epsilon)$$

[4] A star body is a closed set K such that λx is in the interior of K, for all $x \in K$ and $0 \leq \lambda \leq 1$. A full-dimensional convex region is a star body, so Proposition 5 applies to K such that $\mathcal{D} = \mathring{K}$ with our assumptions.

which means that the ratio $\Delta_{\mathbb{Z}}(\lambda K)/\Delta(\lambda K)$ is less than $1+3\epsilon$ when λ is large enough. In other words, the limit of $\Delta_{\mathbb{Z}}(\lambda K)/\Delta(\lambda K)$, when λ grows to infinity, is 1. ⊠

4.4 Lower bounds & Minkowski's first theorem

Remember our first lower bounds in Section 3.2.2. If there is a set S such that $S-S\subseteq\mathcal{D}$, then $\mathrm{size}(\sigma)\geq\mathrm{Card}(S)$ for any $\mathcal{C}$-valid linear mapping. In terms of strictly admissible lattices, this means that for any set S such that $S-S\subseteq K$, $\Delta_{\mathbb{Z}}(K)\geq\mathrm{Card}(S)$. We can make the same argument concerning a 0-symmetric bounded convex body K in $\mathbb{R}^n$. Let $\Lambda\subseteq\mathbb{Z}^n$ be a strictly admissible lattice for K, then 2Λ is strictly admissible for $2K$ and $\mathring{K}-\mathring{K}\subseteq K-K=2K$. Thus, we get $d(2\Lambda)\geq\mathrm{Card}(\mathring{K})$ and $d(\Lambda)\geq\mathrm{Card}(\mathring{K})/2^n$. It follows that

$$\Delta_{\mathbb{Z}}(K)\geq\mathrm{Card}(\mathring{K})/2^n.$$

We can get similar inequalities using volumes, thanks to Minkowski's first theorem (see for example [19]). Indeed, if Λ is a lattice of $\mathbb{R}^n$ and K a 0-symmetric bounded convex body such that $\mathrm{Vol}(K)>2^n d(\Lambda)$, then K contains a nonzero lattice point of Λ. In other words, if Λ is a strictly admissible lattice for K, then $d(\Lambda)\geq\mathrm{Vol}(K)/2^n$. This shows that

$$\Delta_{\mathbb{Z}}(K)\geq\mathrm{Vol}(K)/2^n$$

with strict inequality if K is closed. Note that this last inequality with volumes, when applied to our original allocation problem, i.e., in terms of the set of conflicting indices, gives a lower bound similar to the lower bound corresponding to MAXLIVE. Indeed, let $S=\mathring{K}_S$ be a set of integer points within a polytope K_S such that $K_S-K_S\subseteq K$ where $\mathcal{D}=\mathring{K}$. Then $\Delta_{\mathbb{Z}}(K)\geq\mathrm{Vol}(K_S-K_S)/2^n$ and, because of Brunn-Minkowski theorem (see [13, p. 12]), we get $\Delta_{\mathbb{Z}}(K)\geq\mathrm{Vol}(K_S)$, an inequality similar to the one for MAXLIVE, but with volumes.

Example 2 (Cont'd) For the polytope K of Figure 8, obtained with an approximation of the set DS, the first theorem of Minkowski leads to the inequality $\Delta_{\mathbb{Z}}(K)\geq 2(N-1)/2^2=(N-1)/2$. As mentioned before, this shows that this (apparently slight) over-approximation prevents valid modular allocations with a $O(1)$ memory size. □

For an upper bound, we might expect to adapt the Minkowski-Hlawka theorem (see [14] for an elementary "constructive" proof), which states that if K is any bounded Jordan measurable set in $\mathbb{R}^n$, then $\Delta(K)\leq\mathrm{Vol}(K)$. Unfortunately, the construction needs to "refine" the lattice $\mathbb{Z}^n$ (i.e., it uses lattices for which $\mathbb{Z}^n$ is a sublattice) while, for $\Delta_{\mathbb{Z}}(K)$, we want to restrict to sublattices of $\mathbb{Z}^n$. Thus, it is not clear whether an equivalent of this theorem can be given for $\Delta_{\mathbb{Z}}(K)$. Nevertheless, since $\Delta_{\mathbb{Z}}(K)/\mathrm{Vol}(K)$ is lower bounded by a function that depends on the problem dimension n only, we are going to look for heuristics that build integer lattices Λ, strictly admissible for K, such that the ratio $d(\Lambda)/\mathrm{Vol}(K)$ is upper bounded by a function of n only. This will give us heuristics that are *optimal up to a multiplicative factor*.

4.5 Optimal construction

The number of integer lattices of given determinant is finite (see the following paragraph for proof), and so in principle, $\Delta_{\mathbb{Z}}(K)$ can be found by exhaustive search, assuming the availability of an oracle that determines the strict admissibility for K of a candidate lattice. Though not practical for bodies with very large volumes, this search will give a $\mathcal{C}$-valid linear allocation that requires the least possible storage $\Delta_{\mathbb{Z}}(\mathcal{D})$, and that can be implemented. This is how we found the optimal modular allocation for the detailed example that we study in Section 7.

Indeed, since any integer lattice has a basis whose corresponding matrix is nonnegative, lower triangular, with all components below the diagonal strictly smaller than the diagonal element of the same row, and with a factorization of the determinant on the diagonal (this is the Hermite form we introduced in Section 4.1), we can generate all integer lattices with a given determinant d. For each such lattice, we can check (either with integer linear programming or by solving an integer triangular system, for each fixed element in $\mathring{K}$, if we can enumerate them) that it intersects K only in $\vec{0}$. If we find no strictly admissible lattice with determinant d, we continue the search with the value $d+1$. The procedure will stop with a solution for a determinant less than a linear function of $\mathrm{Vol}(K)$ as the next sections show. The complexity of this procedure depends on the number $H_n(d)$ of lattices of determinant d. If $d = \pi^k$ with π a prime, then there is a closed form: $H_n(d) = \prod_{j=1}^{n-1}(\pi^{k+j}-1)/(\pi^j-1)$; otherwise the number of lattices satisfies the formula $H_n(d) = H_n(p)H_n(q)$ where $d = pq$, p and q relatively prime (see [25, Theorem II.4]).

Example 1 (Cont'd) The body K with $\mathring{K} = DS$ in Figure 7 has a very small volume, equal to 2, which makes the exhaustive search practical. We just have to consider 4 lattices, the lattice generated by a) the vectors $(1,0)$ and $(0,1)$ for the only lattice with determinant 1, b) the vectors $(1,0)$ and $(0,2)$, c) the vectors $(2,0)$ and $(0,1)$, and d) the vectors $(1,1)$ and $(0,2)$. Depending on the parity of N, we find the two optimal allocations, with memory size 2, mentioned before (they both correspond to $Ni+j \bmod 2$), $j \bmod 2$ (Case b and N even) and $i+j \bmod 2$ (Case d and N odd). Applying this principle "blindly" to the body of Figure 8, when N is large, will clearly be very expensive. $\square$

5 Memory allocation heuristics

We present several heuristics for building $\mathcal{C}$-valid linear allocations. As already discussed, we assume that $\mathcal{D}$ is the set of integer points in a 0-symmetric convex body (closed and bounded) $K \subset \mathbb{R}^n$. In addition, for studying the performance of heuristics, we assume that K contains n linearly independent integer points. According to Proposition 4, the problem is equivalent to that of constructing an integer lattice Λ, strictly admissible for K, of smallest possible determinant.

Note also that if $\mathcal{D}$ is a subset of the interior of K, then any strictly admissible lattice Λ is a packing lattice for $K/2$. In this case the question may also be viewed as maximizing the ratio $\mathrm{Vol}(K)/(2^n d(\Lambda))$, which is maximizing the density of the packing $\Lambda + K/2$ over all integer lattices.

The heuristics we propose for finding integer strictly admissible lattices for K are based on a scaling scheme. We start with n independent integer vectors. The corresponding lattice, or its dual set (see the explanation below), is then scaled up, in such a way that is remains integer, and so as to make it strictly admissible. Valid scaling factors are determined by different techniques. In Section 5.2 we use the successive minima of K with respect to $\mathbb{Z}^n$. In Sections 5.3 and 5.4 the scaling factors are computed using fixed or generalized reduced bases [24] in K and its dual body K^*. We propose a polynomial-time heuristic based on LLL lattice basis reduction [22] in Section 5.5.

5.1 Gauge function, dual sets, and successive minima

Let us first recall some notions and fundamental results on convex bodies, dual sets, and successive minima (see for example [13] for details). When $K \subset \mathbb{R}^n$ is a 0-symmetric closed bounded convex body, then

$$F(\vec{x}) = \inf\{\lambda > 0 \mid \vec{x} \in \lambda K\}$$

defines a distance function such that $F(\alpha\vec{x}) = |\alpha| F(\vec{x})$, called the **gauge (or distance) function** of K. The set K is then the set of points $\vec{x}$ such that $F(\vec{x}) \leq 1$. Given a basis $(\vec{a}_i)_{1\leq i\leq n}$ of a lattice $L \subseteq \mathbb{Z}^n$, we let K_i be the projection of K along $\vec{a}_1, \ldots, \vec{a}_{i-1}$ into $\mathrm{Vect}(\vec{a}_i, \ldots, \vec{a}_n)$, where Vect denotes the linear span, and we define for each i and $\vec{x} \in K_i$ the gauge function

$$F_i(\vec{x}) = \inf\{F(\vec{y}) \mid \vec{y} \in \vec{x} + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\}.$$

For an upper bound on the product of the F_i's, we shall use the fact that, for any basis, the following inequality holds [24, Theorem 4]:

$$\prod_{i=1}^{n} F_i(\vec{a}_i)\mathrm{Vol}(K) \leq 2^n. \tag{2}$$

The **dual body** or **polar reciprocal** of K is the set

$$K^* = \{\vec{y} \in \mathbb{R}^n \mid \vec{x}.\vec{y} \leq 1 \text{ for all } \vec{x} \in K\},$$

where . is the inner product of $\vec{x}$ and $\vec{y}$. The set K^* is also a 0-symmetric closed bounded convex body, such that $(K^*)^* = K$. The volumes of K and K^* satisfy (see Theorem 14.4 in [13]):

$$\mathrm{Vol}(K)\mathrm{Vol}(K^*) \geq 4^n/(n!)^2. \tag{3}$$

Using linear programming duality, it is possible to show that

$$F_i(\vec{x}) = \sup\{\vec{x}.\vec{z} \mid \vec{z} \in K^*, \vec{a}_1.\vec{z} = \ldots = \vec{a}_{i-1}.\vec{z} = 0\}. \tag{4}$$

For more details about this result and the functions F_i, see [24] and its annotated version [15]. In particular, following (4) with $(K^*)^* = K$, the gauge function of K^* is $F^*(\vec{y}) = \sup\{\vec{y}.\vec{x} \mid \vec{x} \in K\}$ and

$$F_i^*(\vec{y}) = \sup\{\vec{y}.\vec{z} \mid \vec{z} \in K, \vec{a}_1.\vec{z} = \ldots = \vec{a}_{i-1}.\vec{z} = 0\}. \tag{5}$$

The **dual** of a lattice $L = A\mathbb{Z}^n$ of rank n is the lattice $L^* = (A^{-1})^{\mathrm{tr}}\mathbb{Z}^n$. If A has column vectors $(\vec{a}_i)_{1\leq i\leq n}$, and A^{-1} has row vectors $(\vec{b}_i)_{1\leq i\leq n}$, the gauge functions with respect to the lattice and its dual are related as follows.

Lemma 2 *The gauge functions F_i, with respect to $(\vec{a}_i)_{1\leq i\leq n}$, and F_i^*, with respect to $(\vec{c}_i)_{1\leq i\leq n} = (\vec{b}_{n-i+1})_{1\leq i\leq n}$, satisfy $F_i(\vec{a}_i)F^*_{n-i+1}(\vec{c}_{n-i+1}) = 1$.*

Proof. We have:

$$\begin{aligned} F_i(\vec{a}_i) &= \inf\{F(\vec{y}) \mid \vec{y} \in \vec{a}_i + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\} \\ &= \inf\{F(\vec{y}) \mid \vec{b}_i.\vec{y} = 1,\, \vec{b}_{i+1}.\vec{y} = \ldots = \vec{b}_n.\vec{y} = 0\} \\ &= \inf\{\rho > 0 \mid \vec{y} \in \rho K,\, \vec{b}_i.\vec{y} = 1,\, \vec{b}_{i+1}.\vec{y} = \ldots = \vec{b}_n.\vec{y} = 0\} \\ &= \inf\{\rho > 0 \mid \vec{y} \in K,\, \vec{b}_i.\vec{y} = 1/\rho,\, \vec{b}_{i+1}.\vec{y} = \ldots = \vec{b}_n.\vec{y} = 0\} \\ &= 1/\sup\{\vec{b}_i.\vec{y} \mid \vec{y} \in K,\, \vec{b}_{i+1}.\vec{y} = \ldots = \vec{b}_n.\vec{y} = 0\} \end{aligned}$$

Using (5), the last expression is $1/F^*_{n+1-i}(\vec{c}_{n+1-i})$. ⊠

Given a lattice L, the n **successive minima** of K with respect to L are defined, for $1 \leq i \leq n$, by

$$\lambda_i(K, L) = \inf\{\lambda > 0 \mid \dim(\mathrm{Vect}(\lambda K \cap L)) \geq i\}.$$

In other words, $\lambda_i(K, L)$ is the smallest value λ such that L contains at least i linearly independent points with $F(x) \leq \lambda$. For the following, refer for instance to [13, p. 62].

Theorem 2 (Minkowski's Second Theorem, 1896) *For a 0-symmetric convex body K and a lattice L,*

$$(2^n/n!)d(L) \leq \lambda_1(K,L)\ldots\lambda_n(K,L)\mathrm{Vol}(K) \leq 2^n d(L).$$

As mentioned earlier, we shall restrict our uses of Minkowski's Theorem to the case $L = \mathbb{Z}^n$ ($d(L) = 1$), and simply write $\lambda_i(K)$.

The successive minima $\lambda_i(K^*)$ of K^* with respect to $\mathbb{Z}^n$ are defined similarly, and they satisfy [13, Theorem 14.5]

$$\lambda_i(K^*)\lambda_{n+1-i}(K) \geq 1. \tag{6}$$

5.2 Using the successive minima

We adapt to $\Delta_{\mathbb{Z}}(K)$ a technique that Rogers developed for $\Delta(K)$ (see Theorem 18.1 in [13]). Rogers' approach leads to a noninteger strictly admissible lattice. It is based on a scaling determined by the successive minima of K. We modify the construction to get an appropriate *integer* lattice Λ.

Heuristic 1

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

The next theorem shows that the loss in accuracy is limited; the ratio $d(\Lambda)/\mathrm{Vol}(K)$ remains upper bounded by a function of n only.

Theorem 3 *The lattice Λ computed by Heuristic 1 is a strictly admissible integer lattice for K. Furthermore, if K is a 0-symmetric convex body in $\mathbb{R}^n$, with n linearly independent integer points, and the ρ_i's are the smallest possible powers of 2, then*

$$\lambda_1(K)\ldots\lambda_n(K)d(\Lambda) \leq 2^n, \tag{7}$$

and, consequently,

$$d(\Lambda) \leq n!\,\mathrm{Vol}(K). \tag{8}$$

Proof. We first prove that the construction (the first bullet) is possible. Indeed, if $\rho_i > 1/\lambda_i(K)$, then $\dim(\mathcal{L}_i) \leq i-1$ since $\lambda_i(K)$ is the smallest value such that $\dim(\mathrm{Vect}(K/\rho_i \cap \mathbb{Z}^n)) \geq i$. For the divisibility condition, one can choose ρ_i to be the smallest integer power of 2 such that $\rho_i\lambda_i(K) > 1$. Then ρ_i is a multiple of ρ_{i+1} since $\lambda_i(K) \leq \lambda_{i+1}(K)$. Note that the divisibility condition implies that $\mathcal{L}_i \subseteq \mathcal{L}_{i+1}$. Indeed, if $\vec{x} \in K/\rho_i \cap \mathbb{Z}^n$, then with $m_i = \rho_i/\rho_{i+1}$, which is an integer, $m_i\vec{x} \in K/\rho_{i+1} \cap \mathbb{Z}^n$. It follows that an adequate basis $(\vec{a}_i)_{1\leq i\leq n}$ can be built.

Now, consider the integer lattice Λ with basis $(\rho_1\vec{a}_1,\ \ldots,\ \rho_n\vec{a}_n)$. Let $\vec{x} \in \Lambda$, $\vec{x} = \sum_{j=1}^{i} x_j\rho_j\vec{a}_j$ with $x_1,\ \ldots,\ x_i$ integers, and $x_i \neq 0$. We get $\vec{x}/\rho_i = v_1\vec{a}_1 + \cdots + v_i\vec{a}_i$, where $v_1,\ldots,v_i$ are integers, $v_i \neq 0$, since ρ_i divides all ρ_j, $1 \leq j < i$. Therefore, $\vec{x}/\rho_i \in \mathbb{Z}^n$. Moreover, $\vec{x}/\rho_i \notin K/\rho_i \cap \mathbb{Z}^n$ since $K/\rho_i \cap \mathbb{Z}^n \subseteq \mathcal{L}_i \subseteq \mathrm{Vect}(\vec{a}_1,\ldots,\vec{a}_{i-1})$. Hence, $\vec{x}/\rho_i \notin K/\rho_i$, i.e., $\vec{x} \notin K$. It follows that Λ is strictly admissible for K.

If K contains n linearly independent integer points, then $\lambda_i(K) \leq 1$. Then, if ρ_i is the smallest power of 2 such that $\rho_i \lambda_i(K) > 1$ (which, as we just showed, leads to a valid construction), $\rho_i \lambda_i(K) \leq 2$, and we get:

$$d(\Lambda) \prod_{i=1}^{n} \lambda_i(K) = \prod_{i=1}^{n} \rho_i \lambda_i(K) \leq 2^n$$

Finally, with $\prod_{i=1}^{n} \lambda_i(K)\mathrm{Vol}(K) \geq 2^n/n!$ (Theorem 2), the desired upper bound follows. ⊠

Inequality (7) also gives $\lambda_1(K) \dots \lambda_n(K) \Delta_{\mathbb{Z}}(K) \leq 2^n$. This may be compared to the more general statement of Rogers, that for a 0-symmetric convex body K and a lattice L,

$$\lambda_1(K, L) \dots \lambda_n(K, L) \Delta(K) \leq 2^{\frac{n-1}{2}}.$$

The latter uses an approximation scheme that we have coerced into the integers to obtain the bound with $\Delta_{\mathbb{Z}}(K)$. Note that this upper bound may not be the best possible. In particular, inequality (7) remains valid for arbitrary K. There must be space for improvement to better exploit the fact that K is a 0-symmetric convex body. Also it is maybe possible to improve this scheme by considering non integer ρ_i and a rational basis $(\vec{a}_i)_{1 \leq i \leq n}$, as long as $\rho_i \vec{a}_i$ is integer.

Example 1 (Cont'd) The two successive minima of K (see Figure 7) with respect to $\mathbb{Z}^n$ are $\lambda_1(K) = \lambda_2(K) = 1$. With $\rho_1 = \rho_2 = 2$, we see that with Heuristic 1, we get $\mathcal{L}_1 = \mathcal{L}_2 = \{\vec{0}\}$ since $K/2 \cap \mathbb{Z}^n = \{\vec{0}\}$. Thus, whatever the choice of a basis $(\vec{a}_1, \vec{a}_2)$ of $\mathbb{Z}^n$, the lattice generated by $2\vec{a}_1$ and $2\vec{a}_2$ is strictly admissible. This shows that for any unimodular matrix U, the allocation $\sigma(\vec{x}) = U\vec{x} \bmod \vec{b}$ is valid for $\vec{b} = (2, 2)$. Actually, this is also obvious from the fact that DS contains only primitive vectors while $U\vec{x} \bmod \vec{b} = 0$ iff $\vec{x} = 2U^{-1}\vec{y}$ for some integer vector $\vec{y}$, which is $\ker(\sigma) = 2\mathbb{Z}^2$.

Now, if the set DS is approximated by the polytope K in Figure 8, $\lambda_1(K) = 1/(N-1)$ is reached by the vector $(0, 1)$, and $\lambda_2(K) = 1$ is reached by the vector $(1, 1-N)$. Based on Heuristic 1, we choose the smallest powers of 2, ρ_1 and ρ_2, with $\rho_1 \lambda_1(K) > 1$ and $\rho_2 \lambda_2(K) > 1$, hence $\rho_1 = 2^p \geq N$ and $\rho_2 = 2$. This fixes $\vec{a}_1 = (0, 1)$ in the direction of the first minimum. Then $\vec{a}_2$ is any vector such that $(\vec{a}_1, \vec{a}_2)$ is a basis of $\mathbb{Z}^n$, for example $(1, 0)$. By construction, the lattice generated by $2^p\vec{a}_1$ and $2\vec{a}_2$ is strictly admissible, which corresponds to the mapping $(i \bmod 2, j \bmod 2^p)$. Note that the heuristic imposes $\vec{a}_1 = (0, 1)$. For instance, the mapping $(i \bmod 2^p, j \bmod 2)$ is not valid; one needs to be careful about the unequal extents of the polytope. □

5.3 Heuristics based on the body K

The previous construction based on Rogers' approach scales a basis whose axes depends on the successive minima. For computing scaling directions, the next heuristic is guided instead by the gauge functions F_i of the projections K_i. The functions F_i are based on rational points and thus will capture integer points with less accuracy than the scaling in Heuristic 1 (compare inequality (9) to inequality (8)). The projection approach may be preferable for computational reasons: it does not require the computation of the successive minima, and we shall see that it also always leads to a valid 1D modular mapping. There seems to be no simple mechanism to derive such a 1D mapping from Heuristic 1, which leads in general to an n-dimensional allocation. (About the number of dimensions of the allocation array, see Theorem 1).

The construction below scales n given linearly independent vectors $(\vec{a}_1, \dots, \vec{a}_n)$ – with multiplicative factors ρ_i – so that the new vectors $(\rho_i \vec{a}_i)_{1 \leq i \leq n}$ generate a strictly admissible integer lattice for K.

Heuristic 2

- *Choose a set of* n *linearly independent integer vectors* $(\vec{a}_1, \ldots, \vec{a}_n)$.
- *Compute the quantities* $F_i(\vec{a}_i) = \inf\{F(\vec{y}) \mid \vec{y} \in \vec{a}_i + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})\}$, $1 \le i \le n$.
- *Choose* n *integers* ρ_i *such that* $\rho_i F_i(\vec{a}_i) > 1$.
- *Define* Λ *to be the lattice generated by the vectors* $(\rho_i \vec{a}_i)_{1 \le i \le n}$.

Theorem 4 *The lattice* Λ *built by Heuristic 2 is a strictly admissible integer lattice for* K*. Furthermore, if the vectors* $(\vec{a}_i)_{1\le i\le n}$ *form a basis of* $\mathbb{Z}^n$ *such that* $F_i(\vec{a}_i) \le 1$, $1 \le i \le n$, *then the smallest choice for* $(\rho_i)_{1\le i\le n}$ *leads to* $d(\Lambda)\prod_i F_i(\vec{a}_i) \le 2^n$, *hence*

$$d(\Lambda) \le (n!)^2 \mathrm{Vol}(K). \tag{9}$$

Proof. Let $\vec{x} \in \Lambda$, $\vec{x} = \sum_{j=1}^{i} x_j \rho_j \vec{a}_j$ with $x_1, \ldots, x_i$ integers, and $x_i \neq 0$. Let us check that ρ_i is chosen large enough so that $F(\vec{x}) > 1$, i.e., $\vec{x} \notin K$. By construction, $\vec{x}/(\rho_i x_i)$ belongs to $\vec{a}_i + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})$, therefore $F(\vec{x}/(\rho_i x_i)) \ge F_i(\vec{a}_i)$. Thus, $F(\vec{x}) \ge |x_i|\rho_i F_i(\vec{a}_i) \ge \rho_i F_i(\vec{a}_i) > 1$. This shows that $\vec{x} \notin K$. Since all $\vec{x} \in \Lambda$, $\vec{x} \neq \vec{0}$, have been considered this way, Λ is a strictly admissible integer lattice for K.

Now suppose that the vectors $(\vec{a}_i)_{1\le i\le n}$ form a basis of $\mathbb{Z}^n$, then $d(\Lambda) = \prod_i \rho_i$. Choose the smallest possible integer for ρ_i, i.e., $\rho_i = \lfloor 1/F_i(\vec{a}_i)\rfloor + 1$. If $F_i(\vec{a}_i) > 1$, then $\rho_i = 1$ and it can be ignored in the product. Otherwise $\rho_i \le 1/F_i(\vec{a}_i) + 1 \le 2/F_i(\vec{a}_i)$ and $d(\Lambda) = \prod_{i\in I} \rho_i \le \prod_{i\in I}(2/F_i(\vec{a}_i))$, where I is the set of indices i such that $F_i(\vec{a}_i) \le 1$. When $I = [1 \ldots n]$, it remains to find a lower bound for $\prod_i F_i(\vec{a}_i)$. Lemma 2 gives $d(\Lambda) \le 2^n \prod_{i=1}^n F_i^*(\vec{c}_i)$, and bounding the product of the F_i^*'s with (2), we get $d(\Lambda) \le 4^n/\mathrm{Vol}(K^*)$. The bound (3) for the product of the volume and the dual volume finally leads to $d(\Lambda) \le (n!)^2 \mathrm{Vol}(K)$. ⊠

The performance of Heuristic 2 is not guaranteed if $F_i(\vec{a}_i) > 1$ for some i, in particular if some $F_i(\vec{a}_i)$ is very large, in which case K is very skewed in the basis $(\vec{a}_i)_{1\le i\le n}$. Conversely, if K is not too skewed in the canonical basis $(e_i)_{1\le i\le n}$ (as it is often the case in practice), for example if $\vec{e_i} \in K$, then $F_i(\vec{e}_i) \le F(\vec{e}_i) \le 1$, and Heuristic 2 has guaranteed performance. In the general case, one may use the assumption that K contains n linearly independent integer points $(\vec{x}_i)_{1\le i\le n}$. These points may not form a basis of $\mathbb{Z}^n$ but it is always possible to construct an appropriate basis $(\vec{a}_i)_{1\le i\le n}$ such that $\mathrm{Vect}(\vec{x}_1, \ldots, \vec{x}_i) = \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_i)$ for $1 \le i \le n$. Then, we get $F_i(\vec{a}_i) \le F_i(\vec{x}_i) \le F(\vec{x}_i) \le 1$ and Theorem 4 applies. This is for example the situation corresponding to $F(\vec{x}_i) = \lambda_i(K)$, the successive minima of K with respect to $\mathbb{Z}^n$.

Theorem 1 shows that the number of dimensions of the allocation array is given by the number of non-unit entries in the Smith form of A when $\Lambda = A\mathbb{Z}^n$. Using this result, we can slightly modify Heuristic 2 to get the following.

Corollary 1 *From a strictly admissible lattice* Λ *built by Heuristic 2, one can build a strictly admissible lattice* Λ' *that corresponds to a 1D modular mapping and for which* $\det(\Lambda') = \det(\Lambda)$.

Proof. Given a basis $(\vec{a}_i)_{1\le i\le n}$ of $\mathbb{Z}^n$ and the corresponding ρ_i, consider the lattice Λ' generated by the vectors $(\rho_i\vec{a}_i + \vec{a}_{i-1})_{1\le i\le n}$ with $\vec{a}_0 = \vec{0}$. The same reasoning shows that Λ' is strictly admissible for K because, if $\vec{x} = x_1\rho_1\vec{a}_1 + \sum_{j=2}^{i} x_j(\rho_j\vec{a}_j + \vec{a}_{j-1})$, then $\vec{x} = \sum_{j=1}^{i-1}(x_j\rho_j + x_{j+1})\,\vec{a}_j + x_i\rho_i\vec{a}_i$ and $\vec{x}/(\rho_i x_i)$ belongs to $\vec{a}_i + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1})$. If A is the unimodular matrix whose columns are the $\vec{a}_i$'s, then the matrix for Λ' is $A' = AN$ where $N_{i,i} = \rho_i$ for $1 \le i \le n$, $N_{i,i+1} = 1$ for $1 \le i \le n-1$, and $N_{i,j} = 0$ otherwise. This gives $d(\Lambda') = \prod_i \rho_i = d(\Lambda)$. Furthermore, A is unimodular, hence the smith form of A' is the Smith form of N, which has a unique non-unit entry since, for each i, $1 \leq i < n$, N has a submatrix of size i with determinant one. Finally, according to Theorem 1, the modular mapping associated to A' (as built in Proposition 3) is a one-dimensional mapping. (Note however that the mapping corresponding to Λ and the mapping corresponding to Λ' are *not* equivalent, even though they have same size, because they do not have the same kernel.) $\boxtimes$

The performance of our general scheme relies on small scaling factors for limiting the determinant of the resulting strictly admissible lattice Λ. Heuristic 2 depends on the chosen basis for $\mathbb{Z}^n$, and the actual performance is related to the fact that $F_i(\vec{a}_i)$ can be lower bounded, so as to get an upper bound for $\prod_i(\lfloor 1/F_i(\vec{a}_i)\rfloor + 1)$. The **generalized basis reduction** of [24] provides different lower bounds for the norms (where the norms are defined with respect to the gauge functions corresponding to K). In addition, we now show that it leads to a performance guarantee even when the $F_i(\vec{a}_i)$ are not all smaller than 1 (provided that, as in our case, ρ_i can be upper bounded for each i).

Given a basis $(\vec{a}_i)_{1\leq i\leq n}$ for $\mathbb{Z}^n$, we first apply the algorithm of [24] for computing a reduced basis $(\vec{r}_i)_{1\leq i\leq n}$, which is then used in Heuristic 2. According to [24, Theorem 3], the reduced basis, defined for given ϵ, $0 < \epsilon < 1/2$, is such that

$$\lambda_i(K)(1/2-\epsilon)^{i-1} \leq F_i(\vec{r}_i) \leq \lambda_i(K)(1/2-\epsilon)^{i-n} \tag{10}$$

Hence, for example with $\epsilon = 1/4$, the resulting allocation has memory size

$$\begin{array}{rcl} d(\Lambda) & \leq & \prod_{i=1}^n\left(\frac{1}{F_i(\vec{r}_i)}+1\right) \leq \prod_{i=1}^n\left(\frac{4^{i-1}}{\lambda_i(K)}+1\right) \\ & \leq & \prod_{i=1}^n\left(\frac{4^{i-1}+1}{\lambda_i(K)}\right) \leq \frac{(n+1)2^{n(n-1)}}{\prod_{i=1}^n \lambda_i(K)} \leq 2^{n(n-2)}(n+1)!\mathrm{Vol}(K), \end{array} \tag{11}$$

the third inequality because we have assumed that K contains n linearly independent integer points (i.e., $\lambda_i(K) \leq 1$ for all i), the last inequality is from Theorem 2.

It can be shown that this technique will work even if K is a polytope in $\mathbb{Z}^n$ of dimension q that contains only $p \leq q$ linearly independent integer points. The performance will then depend both on p and q. Indeed, we can first identify the smallest vector space V of dimension q that contains K, thanks to rational linear programming. Then K, expressed as a rational polyhedron in this vector space, has a positive volume (even though it may contain only $p < q$ linearly independent integer points), and we can now apply Lovasz and Scarf technique to find a generalized reduced basis. Then, we get:

$$\begin{array}{rcl} d(\Lambda) & \leq & \prod_{i=1}^q\left(\frac{4^{i-1}}{\lambda_i(K)}+1\right) \leq \prod_{i=1}^p\left(\frac{4^{i-1}+1}{\lambda_i(K)}\right)\prod_{i=p+1}^q(4^{i-1}+1) \\ & \leq & (q+1)2^{q(q-1)-p}p\,!\mathrm{Vol}(K') \leq (q+1)2^{q(q-1)}p\,!\Delta_{\mathbb{Z}}(K) \end{array}$$

where K' is the intersection of K with the smallest vector space that contains all integer points in K (we have $\lambda_i(K) = \lambda_i(K')$ for $1 \leq i \leq p$, and $\Delta_{\mathbb{Z}}(K) \geq \Delta_{\mathbb{Z}}(K') \geq \mathrm{Vol}(K')/2^p$). In other words, if the dimension q of K is strictly more than the dimension p of K' then, since the generalized reduced basis does not exactly identify K', we may loose a factor that depends on q and not only on p (however, in practice, for our allocation problem, K is more likely to have integer vertices, or we may identify K' first, thus $p = q$).

Example 1 (Cont'd) Consider again the body K in Figure 7, with $\lambda_1(K) = \lambda_2(K) = 1$, reached by the 2 linearly independent vectors $(0,1)$ and $(1, 1-N)$. Using Heuristic 2 for the basis $\vec{a}_1 = (1,0)$, $\vec{a}_2 = (0,1)$ (in this order) leads to $F_1(\vec{a}_1) = N > 1$ (thus $\rho_1 = 1$) and $F_2(\vec{a}_2) = 1/(N-1)$ (thus

$\rho_2 = N$), for a memory of size N. The chosen basis leads to a total scaling factor $\rho_1\rho_2 = N$, which is clearly too big. Precisely, since $F_1(\vec{a}_1) \geq 1$, Heuristic 2 has no guaranteed performance for this particular basis.

The generalized reduced basis approach indicates that the choice of $\vec{a}_2$ leads to a too small norm $F_2(\vec{a}_2)$. In particular, we see from (10) that the basis is not reduced since $F_2(\vec{a}_2) \ll \lambda_2(K)(\frac{1}{2}-\epsilon) = \frac{1}{2}-\epsilon$. With the basis in the opposite order (what the generalized basis reduction would do), we get $\rho_1 = 2$, then $\rho_2 = 2$, for a lattice with determinant 4 (depicted in Figure 7 as grey points). The corresponding modular mapping is $(i \bmod 2, j \bmod 2)$. The lattice Λ', defined for the 1D mapping of Corollary 1, is the lattice generated by $2(0,1) = (0,2)$ and $2(1,0)+(0,1) = (2,1)$, which corresponds to the mapping $i + 2j \bmod 4$. □

5.4 Heuristics based on the body K^*

Instead of working in the space of admissible lattices for K, we shall see that working with the dual body K^* amounts to working directly in the space of modular mappings $(M, \vec{b})$ whose kernels are strictly admissible for K. We consider an approximation scheme dual to Heuristic 2 in the sense that it uses the functions F_i^* (see (5)) instead of the functions F_i. It turns out that this dual mechanism is nothing but the successive moduli principle used by Lefebvre and Feautrier [21] (see Section 2.2), generalized to any set of n linearly independent integer vectors.

Heuristic 3

- *Choose a set of n linearly independent integer vectors $(\vec{c}_1, \ldots, \vec{c}_n)$.*
- *Compute the quantities $F_i^*(\vec{c}_i) = \sup\{\vec{c}_i.\vec{z} \mid \vec{z} \in K, \vec{c}_1.\vec{z} = \ldots = \vec{c}_{i-1}.\vec{z} = 0\}$, $1 \leq i \leq n$.*
- *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 $b_i = \rho_i$.*

Note that the gauge function $F^*(\vec{y}) = \sup\{\vec{y}.\vec{x} \mid \vec{x} \in K\}$ of the dual K^* is half of the width of K (or equivalently the width of $K/2$) in the direction $\vec{y}$. More generally, $F_i^*(\vec{y})$ with respect to $(\vec{c}_i)_{1\leq i\leq n}$ is half the width of K_i in the direction of $\vec{y}$.

Theorem 5 *The kernel Λ of the modular mapping $(M, \vec{b})$ built by Heuristic 3 is a strictly admissible integer lattice for K. If $(\vec{c}_i)_{1\leq i\leq n}$ is a basis of $\mathbb{Z}^n$, the smallest choice for $(\rho_i)_{1\leq i\leq n}$ leads to $d(\Lambda) = \prod_{i=1}^n(\lfloor F_i^*(\vec{c}_i)\rfloor + 1)$. Furthermore, if $F_i^*(\vec{c}_i) \geq 1$, i.e., the "successive widths" of K in the directions $\vec{c}_i$ are more than 1, then, $d(\Lambda) \leq 2^n \prod_i F_i^*(\vec{c}_i)$, hence*

$$d(\Lambda) \leq (n!)^2 \operatorname{Vol}(K). \tag{12}$$

Proof. The statement is dual to the one of Theorem 4 and we could use the same proof. Nevertheless, to better understand the heuristic of Lefebvre and Feautrier [21], which proceeds similarly (but with the particular choice of the canonical basis in which K was defined), here is a direct proof of its validity similar to the proof we used in Section 2.2 for the successive moduli approach.

We show that $\vec{0}$ is the unique integer point in K whose image by the allocation function is 0. Let $\vec{x} \in K$ with $M\vec{x} \bmod \vec{b} = 0$. We have $\vec{c}_1.\vec{x} \bmod b_1 = 0$, but $b_1 = \rho_1 > F_1^*(\vec{c}_1) = \sup\{\vec{c}_1.\vec{x} \mid \vec{x} \in K\} = \sup\{|\vec{c}_1.\vec{x}| \mid \vec{x} \in K\}$ because K is 0-symmetric. Thus, $\vec{c}_1.\vec{x} = 0$. Then, considering $\vec{c}_2.\vec{x} \bmod b_2 = 0$ and $b_2 = \rho_2 > F_2^*(\vec{c}_2) = \sup\{|\vec{c}_2.\vec{x}| \mid \vec{x} \in K,\ \vec{c}_1.\vec{x} = 0\}$, we get $\vec{c}_2.\vec{x} = 0$. Continuing this process, we get $\vec{c}_i.\vec{x} = 0$ for all i. Since the vectors $\vec{c}_i$ are n linearly independent vectors, this implies $\vec{x} = 0$, which proves that the kernel of $(M, \vec{b})$ is strictly admissible for K.

From Theorem 1, we can show that the determinant of the corresponding lattice Λ divides the product of the components of $\vec{b}$, thus $d(\Lambda) \leq \prod_i \rho_i$ (with equality if $(\vec{c}_i)_{1\leq i\leq n}$ is a basis of $\mathbb{Z}^n$). With the smallest possible ρ_i, we get $d(\Lambda) \leq \prod_i(\lfloor F_i^*(\vec{c}_i)\rfloor + 1)$. Finally, when $F_i^*(\vec{c}_i) \geq 1$, the latter bound implies $d(\Lambda) \leq 2^n \prod_{i=1}^n F_i^*(\vec{c}_i)$, and we conclude as for the proof of Theorem 4. ⊠

As done for Heuristic 2 (see the discussion after Theorem 4 and Corollary 1), we can discuss the performance of Heuristic 3 for particular sets $(\vec{c}_i)_{1\leq i\leq n}$. But note that since we work in the dual, inequalities have to be used in the opposite direction, i.e., the performance now relies on the fact that $F_i^*(\vec{c}_i)$ can be upper bounded, so as to get an upper bound for $\prod_i(\lfloor F_i^*(\vec{c}_i)\rfloor + 1)$. Hence, somehow, short vectors with respect to the dual should be preferred. One may use for instance the fact that K contains n linearly independent integer vectors. This gives $\lambda_i(K) \leq 1$ and $\lambda_i(K^*) \geq 1$ for all i using (6). If we consider n linearly independent integer vectors $\vec{c}_i$ such that $F^*(\vec{c}_i) = \lambda_i(K^*)$, then $F_i^*(\vec{c}_i) \leq \lambda_i(K^*)$, and we may not be in the case where $F_i^*(\vec{c}_i) \geq 1$. Nevertheless, we get $d(\Lambda) \leq \prod_i(\lambda_i(K^*) + 1) \leq 2^n \prod_i \lambda_i(K^*)$. [5] Using Theorem 2 for bounding the latter product, and (3) to introduce the volume of K, we finally get $d(\Lambda) \leq (n!)^2\mathrm{Vol}(K)$. Hence the performance of Heuristic 3 is guaranteed without the norm condition for a special basis. The same study can be done with a Korkine-Zolotarev [24] basis for K^* since, for such a basis, $F_i^*(\vec{c}_i) \leq \lambda_i(K^*)$. A generalized reduced basis for K^* would also be appropriate for bounding $d(\Lambda)$.

Example 1 (Cont'd) Working in the dual K^* of K amounts to compute the successive width of K depicted in Figure 7. The width of $K/2$ in the direction of $\vec{c}_1 = (1, 0)$ is $F^*(\vec{c}_1) = F_1^*(\vec{c}_1) = \lambda_1(K^*) = 1$, thus $\rho_1 = 2$. Then, for $\vec{c}_2 = (0, 1)$, we get $F_2^*(\vec{c}_2) = \lambda_2(K^*) = 1$, leading to the valid mapping of size 4, $(i \bmod 2, j \bmod 2)$, already found Page 31 with Heuristic 2.

Considering the canonical basis in the opposite order, i.e., with $\vec{c}_1 = (0, 1)$ and $\vec{c}_2 = (1, 0)$ leads to $F^*(\vec{c}_1) = N - 1$, hence $\rho_1 = N$, which is clearly too big. Precisely, since $F_2^*(\vec{c}_2) \leq 1$, Heuristic 3 has no guaranteed performance for this particular basis. But we see from (10) that the basis is not reduced since $F_1(\vec{c}_1) \gg \lambda_1(K^*)/(\frac{1}{2} - \epsilon) = \frac{2}{1-\epsilon}$. It can be reduced to $\vec{r}_1 = \vec{c}_1 + N\vec{c}_2 = (N, 1)$ with $F^*(\vec{r}_1) = 1$, for a scaling factor 2. Then, with $\vec{r}_2 = \vec{c}_2$, we get $F_2^*(\vec{r}_2) < 1$ for a scaling factor 1, which leads to the mapping $(Ni + j \bmod 2, i \bmod 1)$, i.e., the optimal and 1D mapping $Ni + j \bmod 2$. □

As for Heuristic 2 in Corollary 1, we can slightly modify Heuristic 3 to define a valid one-dimensional mapping. In other words, this is a second (although equivalent to the mechanism of Corollary 1) guaranteed mechanism for deriving valid one-dimensional modular mappings.

Corollary 2 *If $(M, \vec{b})$ is a valid modular mapping built by Heuristic 3, then the modular mapping defined by*

$$(\vec{c}_1 + b_1\vec{c}_2 + b_1b_2\vec{c}_3 + \ldots + b_1 \cdots b_{n-1}\vec{c}_n).\vec{i} \bmod \prod_{i=1}^{n} b_i$$

is a valid 1D mapping of same size.

Proof. The validity of this mapping can be checked as follows: consider $\vec{i} \in \mathring{K}$ such that $\vec{m}.\vec{i} \bmod \prod_i b_i = 0$ where $\vec{m} = \vec{c}_1 + b_1\vec{c}_2 + \ldots + b_1 \cdots b_{n-1}\vec{c}_n$. Then $\vec{m}.\vec{i} \bmod b_1 = 0$, i.e., $\vec{c}_1.\vec{i} \bmod b_1 = 0$. We conclude as in Theorem 5 that $\vec{c}_1.\vec{i} = 0$. Then we get $b_1(\vec{c}_2 + \ldots + b_2 \cdots b_{n-1}\vec{c}_n).\vec{i} \bmod \prod_i b_i$, thus $(\vec{c}_2 + \ldots + b_2 \cdots b_{n-1}\vec{c}_n).\vec{i} \bmod \prod_{i=2}^n b_i$, etc. ⊠

[5] Note that, if we define Λ with $\rho_i = \lfloor F^*(\vec{c}_i)\rfloor + 1$ instead of $\rho_i = \lfloor F_i^*(\vec{c}_i)\rfloor + 1$, the upper bound for $d(\Lambda)$ remains true and, this time, the lattice defines a *bounding box* for K, but this is for a very particular basis.

5.5 A polynomial-time heuristic using LLL basis reduction

Each of the previous heuristics produces lattices with bounded determinant. Unfortunately, none of them has fully polynomial complexity. For the sake of completeness, we now give a polynomial-time heuristic based on Lenstra-Lenstra-Lovász (LLL) basis reduction [22] (see also for example [23,10]).

The idea is to approximate K by an ellipsoid for which the generalized basis reduction – used in Section 5.3 for the bound (11) – is equivalent to the LLL reduction. An ellipsoid $\mathcal{E}$ is built such that $\mathcal{E}/n \subset K \subset \mathcal{E}$ (see [23]) (the factor n can be reduced to $\sqrt{n+1}$ if K is a 0-symmetric polytope), then the ellipsoid is transformed into the unit ball by an affine transformation. The same transformation applied to $\mathbb{Z}^n$ gives an associated lattice L. We now have an Euclidean norm and we can apply the LLL basis reduction to find a reduced basis for L. Following Heuristic 2, which consists in scaling the reduced basis vectors by appropriate factors, gives a sublattice $\Lambda_{\mathcal{B}}$ of L, strictly admissible for the unit ball $\mathcal{B}$. Going back to the original body, we will have a strictly admissible integer lattice $\Lambda_{\mathcal{E}}$ for the ellipsoid $\mathcal{E}$, thus for K. This shows the link between strictly admissible integer lattices and approximations with the LLL basis reduction. The following theorem formalizes this approximation principle and gives its performance.

Theorem 6 *If K is a 0-symmetric bounded convex body, then in polynomial time one can compute a lattice $\Lambda \subseteq \mathbb{Z}^n$ such that*

$$d(\Lambda) \leq 2^{n(n+3)/4} n^n \, Vol(K). \tag{13}$$

Proof. Let L be a lattice in $\mathbb{R}^n$, $(\vec{b}_i)_{1\leq i\leq n}$ a basis of L with Gram-Schmidt orthogonalization $(\vec{b}_i^*)_{1\leq i\leq n}$. Define integers ρ_i such that $\rho_i > 1/||\vec{b}_i^*||$. It is easy to see that the sublattice $\Lambda_{\mathcal{B}}$ of L, with basis $(\rho_i\vec{b}_i)_{1\leq i\leq n}$, is strictly admissible for the unit ball $\mathcal{B}$ in $\mathbb{R}^n$, with determinant $d(\Lambda_{\mathcal{B}}) = d(L)\prod_i \rho_i$. Furthermore, if $(\vec{b}_i)_{1\leq i\leq n}$ is a LLL reduced basis for L and if ρ_i is as small as possible, then $\rho_i \leq 1/||\vec{b}_i^*||+1$, which implies $\rho_i \leq 2^{(i-1)/2}/\lambda_i(\mathcal{B}, L)+1$ because of properties of the LLL reduced basis. Assuming again that $\mathcal{B}$ contains n linearly independent points of L, $\lambda_i(\mathcal{B}, L) \leq 1$ and we get $\rho_i \leq 2^{(i+1)/2}/\lambda_i(\mathcal{B}, L)$. Since $\prod_i \lambda_i(\mathcal{B}, L) \geq d(L)$ (see Theorem 18.4 in [13]), we finally get $d(\Lambda_{\mathcal{B}}) \leq \prod_{i=1}^n 2^{(i+1)/2} = 2^{n(n+3)/4}$.

Now, consider a 0-symmetric bounded convex body: one can build in polynomial time an ellipsoid $\mathcal{E}$ such that $\mathcal{E}/n \subset K \subset \mathcal{E}$ (see [23]) (the factor n can be reduced to $\sqrt{n+1}$ if K is a 0-symmetric polytope). Build an affine transformation f such that $f(\mathcal{E}) = \mathcal{B}$ and apply the previous technique to $L = f(\mathbb{Z}^n)$ to get a sublattice $\Lambda_{\mathcal{B}}$ of L, strictly admissible for $\mathcal{B}$. The lattice $\Lambda_{\mathcal{E}} = f^{-1}(\Lambda_{\mathcal{B}})$ is the desired sublattice of $\mathbb{Z}^n$, strictly admissible for $\mathcal{E}$ and thus for K. Furthermore, $d(\Lambda_{\mathcal{E}}) = \mathrm{Vol}(\mathcal{E})d(\Lambda_{\mathcal{B}})$. Putting it all together, we get $d(\Lambda_{\mathcal{E}}) \leq 2^{n(n+3)/4}n^n\mathrm{Vol}(K)$, which is the desired result with $\Lambda = \Lambda_{\mathcal{E}}$. We can replace n^n by $(n+1)^{n/2}$ if K is a 0-symmetric polytope. ⊠

6 General discussion

All the algorithms we have studied proceed through the solution of two subproblems. They choose either a linearization or of a working basis; they also choose a scaling, i.e., a vector of moduli.

6.1 The choice of a working basis

Our main contribution is to show that the size of the allocation we derive can be guaranteed when the allocation is built from a particular choice of a working basis. For a well-chosen basis, the guarantee is an upper bound $c_n\mathrm{Vol}(K)$ for the allocation size $d(\Lambda)$, where c_n depends only on the dimension n (and not on K). Conversely, a bad choice of linearization or basis can lead to an allocation whose size is arbitrarily large with respect to the volume of K.

In our best performance bound, $c_n = n!$ (see Theorem 3). The corresponding heuristic, on both Example 1 and Example 3 below, achieves memory size $d(\Lambda) \leq c_2 \mathrm{Vol}(K) = 4$, while, on Example 3, the De Greef– Catthoor–De Man (canonical) linearization family, and Lefebvre-Feautrier's basis choice, lead respectively to memory of order N^2 and N.

Example 3 To make the Lefebvre–Feautrier heuristic lose an order of magnitude, it is sufficient to "turn" the body of Figure 7, i.e., to apply a change of basis so that the canonical basis is inadequate. Consider a code similar to the code of Example 1. Iteration (i, j) writes to the array location $A(i, j)$, but with the iteration domain represented on the left of Figure 11 and traversed sequentially with successive diagonals (from bottom-left to upper-right). This corresponds to the multi-dimensional schedule $(i - j, i)$. If a second identical loop, traversed the same way but one clock-cycle later, uses the results of the first loop, we get the set DS represented in Figure 11, with vertices $(1, 1)$, $(-1, -1)$, $(N - 1, N)$, and $(1 - N, -N)$ (to build DS, just consider that any two successive points in the schedule of the first loop are conflicting).

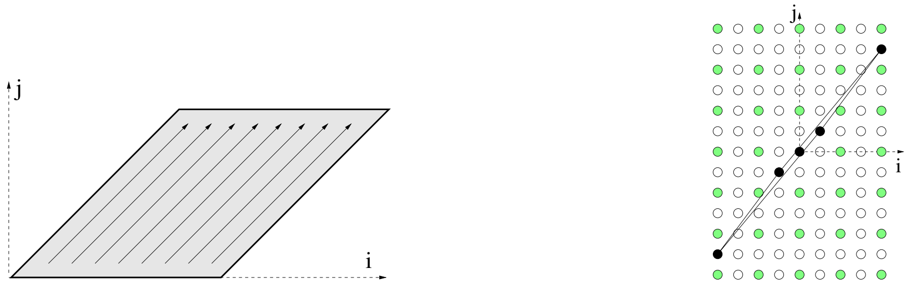

Figure 11: Iteration domain for Example 3 and corresponding set DS.

For this example, the Lefebvre–Feautrier heuristic uses the canonical basis $\{(1, 0), (0, 1)\}$), and chooses $\rho_1 = N + 1$, $\rho_2 = 1$, with a corresponding memory size of order N. Considering the basis in the opposite order does not help ($\rho_1 = N$, $\rho_2 = 1$). Similarly, the De Greef–Catthoor–De Man technique only considers the mappings $\pm Ni \pm j$ and $\pm Nj \pm i$, and all of them have a maximal conflicting distance of order N^2, two orders of magnitude worse than the optimal. $\square$

In addition to showing that the basis plays a key role, our study reveals some essential properties of good linearizations and working bases. To make this explicit, we now discuss the relation between good choices and the geometry of the body K (or equivalently of K^*), under the assumption that K contains n linearly independent integer points.

Regarding bases, we identified sufficient conditions for obtaining a performance guarantee in terms of either K or its dual: the guarantee holds if for all i, $F_i^*(\vec{c}_i)$ is not too large, or $F_i(\vec{a}_i)$ is not too small. One may satisfy this criterion by choosing $(\vec{c}_i)_{1 \leq i \leq n}$ such that $F_i^*(\vec{c}_i) \geq 1$ for all i because, in that case, since there exists a function f_n depending only on n such that $\prod_{1 \leq i \leq n} F_i^*(\vec{c}_i) \leq f_n \mathrm{Vol}(K)$ (using (2) and (3)), it follows that for all i, $F_i^*(\vec{c}_i) \leq f_n \mathrm{Vol}(K)$. Heuristic 3 (Section 5.4) and its dual, Heuristic 2 (Section 5.3) are provably good when we choose the basis following this criterion. Alternatively, if one follows directions corresponding to the successive minima of K^* (see Section 5.4 with the choice $F^*(\vec{c}_i) = \lambda_i(K^*)$) or directions of a reduced basis, the gauge functions $F_i^*(\vec{c}_i)$ of the projections may be directly bounded. The same type of analysis can be done with K directly.

Intuitively, our K^* criterion means that the scaling directions should be such that the corresponding widths of K are balanced. In other words, the polytope should contain integer points in the chosen directions. In particular, a direction in which K is too thin may indicate that it does not contain integer points in this direction. This is typically a loss of guarantee coming from a i_0 such that $F^*_{i_0}(\vec{c}_{i_0}) \ll 1$. Note that the basis $(\vec{c}_i)_{1\leq i\leq n}$ built in Section 5.4 (or $(\vec{a}_i)_{1\leq i\leq n}$ in Section 5.3) is *ordered*. The gauge functions $F^*_i(\vec{c}_i)$ of the projections depend in general on the order in which they are successively computed. Hence, as our examples illustrate (and recall the $2^n n!$ candidate linearizations in the De Greef–Catthoor–De Man heuristic) permuting the basis can improve performance.

At the cost of finding either the successive minima or a generalized reduced basis, we can guarantee an adequate choice of linearization or basis. Applying this strategy to the body of Example 3 overcomes the difficulties that lead other heuristics to err badly and allocate N or N^2 memory; we allocate $O(1)$. This illustrates the key thing that we have shown in this paper: that by carefully choosing the working basis, we have made the modular allocation approach robust.

It may, in some circumstances, be prohibitively expensive to rely on successive minima or generalized basis reduction. It is important, therefore, to point out that previous, fast heuristics can lead to satisfying allocations. In Section 2, we have seen that these heuristics choose bases (or linearizations) related either to the array indices (De Greef, Catthoor, and De Man), to the loop indices (Lefebvre and Feautrier), or to the scheduling function (Quilleré and Rajopadhye). With our framework, we can analyze these different choices. In particular, the performance is guaranteed as soon as the choice is adequate with respect to K. For many hand-written programs encountered in practice, access functions to arrays are simple, scheduling functions (i.e., loop transformations) are not too skewed, and the (possibly sub-)domains where iterations write in memory are well-shaped; therefore the set of conflicting differences DS is not too skewed with respect to the basis given by the array indices, the loop indices, or the schedule.

When we use the iteration vectors as indices for the mapping, the fact that arrays are indexed with skewed access functions is irrelevant, since the original arrays are simply not considered. So when loops are scheduled with a multi-dimensional sequential schedule $\theta(\vec{i}) = (\vec{c}_1.\vec{i}, \ldots, \vec{c}_n.\vec{i})$, one can expect the vectors $\vec{c}_i$ to be a good basis for a heuristic in K^*. In this case, it is often true that all $F^*_i(\vec{c}_i)$ are equal to 0 for the first dimensions, then are larger than 1 for the remaining dimensions. Indeed, as Quilleré and Rajopadhye observed, if the first conflicting difference has depth d, i.e., $d-1$ leading zeros, the set of conflicting differences DS is flat for the first d dimensions, and contains some nonzero elements in all remaining dimensions (except if the "writing" operations belong to a very skewed subspace). In that case, our study shows that we guarantee performance by mixing the techniques of Quilleré and Rajopadhye – for choosing the adequate basis – and of Lefebvre and Feautrier – for choosing the modulus vector. For more complex cases (i.e., when the writing subdomain is skewed with respect to the schedule, or when the schedule itself is a skewed linearization of the iteration domain), our approach first identifies the subspace in which DS lies, then applies Heuristic 1, 2, or 3.

Example 3 (Cont'd) For the set DS of Figure 11, one may follow the schedule, i.e., consider the change of basis $\{\vec{c}_1, \vec{c}_2\} = \{(1,-1), (1,0)\}$. We indeed retrieve the set DS of Figure 7 and we get the allocation $(i - j \bmod 2, i \bmod 2)$. As a matter of fact, when using Heuristic 1) $\rho_1 = \rho_2 = 2$ is acceptable whatever the basis. Thus, the sublattice depicted as grey points in Figure 11 is also valid. $\square$

6.2 Linearizations and moduli

The previous discussion argues about the choice of a working basis or of a particular linearization. What about the way moduli are computed? Heuristic 1 first computes the moduli (the ρ_i's), then a suitable basis for these moduli. The principle is inverted in Heuristic 2 (and equivalently Heuristic 3): the basis is selected first, and the moduli are chosen with respect to this basis. The technique of De Greef, Catthoor, and De Man is also slightly different, in that they work with a particular linearization (which is a one-to-one mapping of the full array space) and compute a single modulus. When this linearization is good, they have an attractive result since, working in one dimension, they need only one modulus. It is a reasonable strategy, then, to first try to reduce the dimension of the problem, then compute the moduli. This is demonstrated by the example of Section 7.

One-dimensional modular mappings have merit in their own right. Machines of course require one-dimensional memory addresses, so any modular multi-dimensional mapping must be followed by a linearization. Moreover, modulo operations and integer divides are quite expensive and address generation is a significant cost in real computation. One-dimensional modular mappings are therefore preferable to those that require more than one modulo operation. We proved (Corollaries 1 and 2) that we can modify any mapping obtained via successive moduli to create a one-dimensional modular mapping with identical memory size. If we don't make this preliminary change to the mapping and we blindly linearize the multi-dimensional array of size $\prod_i b_i$, then, for example, from a mapping $(i, j) \mapsto (i \bmod b_1, j \bmod b_2)$ obtained by the successive moduli approach, we get a linearized access to memory of the form $(i, j) \mapsto \text{base\_address} + (i \bmod b_1) + b_1(j \bmod b_2)$ instead of the simpler form $(i, j) \mapsto \text{base\_address} + (i + b_1 j) \bmod b_1 b_2$. Also, in all heuristics we developed, we can choose moduli that are powers of 2, avoiding integer division and remainder using bit operations.

6.3 Arbitrary sets

The following example shows that, even for simple codes, it may happen that the exact set of conflicting differences DS is not equal to the set of integer points in its convex hull. We can still use Heuristic 1 (valid for any set) to get a strictly admissible lattice for DS. We can also choose a particular basis $(\vec{c}_i)_{1 \le i \le n}$ and apply Heuristic 3, by computing the successive $F_i^*(\vec{c}_i)$, defined as $F_i^*(\vec{c}_i) = \sup\{|\vec{c}_i.\vec{x}| \mid \vec{x} \in DS, \vec{c}_1.\vec{x} = \ldots = \vec{c}_{i-1}.\vec{x} = 0\}$. For this example, Heuristic 1 always gives a good mapping, while Heuristic 3 gives a good mapping only if we choose the basis given by the schedule. In general, even if these heuristics derive valid mappings (or strictly admissible lattices), we do not know their performance for general sets.

Consider the code of Example 1, but traversed with the schedule $\theta(i, j) = (i + j, j)$ (as we did for Example 2, see the new loop bounds on the code of Figure 5) for the first loop, the second loop starting one clock cycle later. Figure 12 gives the set DS in the original basis (on the left) and in the basis given by the schedule (on the right): DS contains $(1, -1)$, all vectors $(p, p-1)$ for $0 \le p < N$, and their opposites. Heuristic 1 shows that, in any basis, one can choose the modulus vector $(2, 2)$ since $\dim(DS) = 2$, but $\dim(\lambda DS) = 0$ for $\lambda < 1$ (thus $\lambda_1(DS) = \lambda_2(DS) = 1$). Therefore, for example, the mapping $(i \bmod 2, j \bmod 2)$ is valid (its kernel is depicted in grey). The successive moduli approach applied in the original basis leads to $b_1 = N$, then $b_2 = 1$, for a memory of size N. In the basis of the schedule, we get $b_1 = b_2 = 2$, for the mapping $(i + j \bmod 2, j \bmod 2)$, which is equivalent to $(i \bmod 2, j \bmod 2)$.

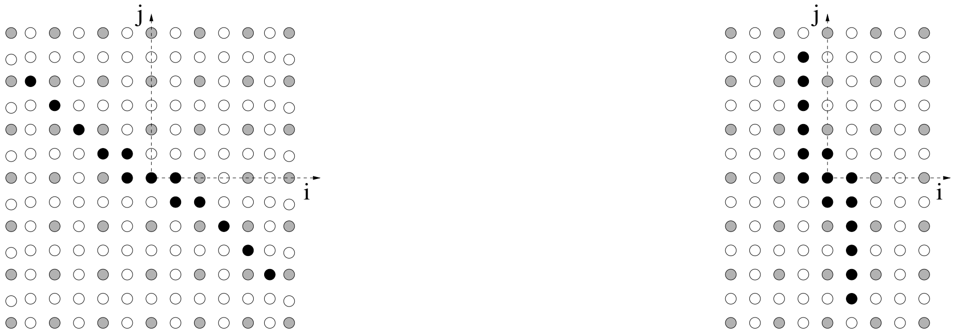

Figure 12: The set DS in the original basis and in the basis given by the schedule.

7 A more detailed case study

We illustrate the interest of modular mappings with a more involved and fully detailed example, in 4 dimensions, similar to the DCT benchmark, but where some details are abstracted so as to make the discussion simpler. The code accesses a 4-dimensional array $A(b_r, b_c, r, c)$, in two pipelined communicating loops, the first one writing to each "row" $A(b_r, b_c, r, *)$ of the array successively, the second reading each "column" $A(b_r, b_c, *, c)$ successively (see code hereafter). Here, to make things simpler, we assume that each operation of S writes to all elements of a row in "parallel", i.e., at the same "macro-time" $(64 \times b_r + b_c) \times 8 + r$ (in other words, the loop is scheduled sequentially as it is written), and each operation of T reads all elements of a column at macro-time $(64 \times b_r + b_c) \times 8 + c + \rho$, where ρ is such that dependences are respected. (In a fully detailed implementation, S and T are formed of several "micro-statements", each one accessing 1 element of A instead of 8. These micro-statements can be software pipelined as done in [17], taking into account the available resources, in particular load-store units, possibly leading to different ρ's for each micro-statement.)

```
DO b_r = 0, 63
  DO b_c = 0, 63
    DO r = 0, 7
      S: A(b_r, b_c, r, *) = ...
    ENDDO
  ENDDO
ENDDO

DO b_r = 0, 63
  DO b_c = 0, 63
    DO c = 0, 7
      T: ... = A(b_r, b_c, *, c)
    ENDDO
  ENDDO
ENDDO
```

For all dependences to be respected, ρ must be such that $(64 \times b_r + b_c) \times 8 + c + \rho \geq (64 \times b_r + b_c) \times 8 + r + 1$ (in a more accurate model again, the delay 1 may be replaced by a larger quantity, depending on the delay for accessing the communicating buffer where the values created by the first loop are going to be stored), i.e., $\rho \geq r - c + 1$ for all $0 \leq r, c < 8$. For the rest of this case study, we pick the smallest possible value for ρ, i.e., $\rho = 8$.

We now need to decide how we want to represent the values that are going to be stored in the intermediate buffer. We can identify each operation of S by the loop indices (b_r, b_c, r) of the surrounding loops, but as each operation of S writes 8 values, we need an extra index to distinguish the different values. In other words, we identify each created value by its indices (b_r, b_c, r, c), as in the original array. As in Example 1, reasoning with loop or array indices is similar (except that we need an extra dimension for loop indices) because the array accesses are aligned with the loop indices.

We can now define the polytope K, with respect to the representation (b_r, b_c, r, c), as the set of all $(\delta_{br}, \delta_{bc}, \delta_r, \delta_c)$ such that:

$$\begin{cases} \delta_{br} = b_r - b'_r,\ \delta_{bc} = b_c - b'_c,\ \delta_r = r - r',\ \delta_c = c - c' \\ 0 \leq b_r, b'_r, b_c, b'_c \leq 63,\ \ 0 \leq r, r', c, c' \leq 7, \\ 64 \times 8 \times \delta_{br} + 8 \times \delta_{bc} + r - c' \leq \rho, \\ 64 \times 8 \times \delta_{br} + 8 \times \delta_{bc} + c - r' \geq -\rho. \end{cases}$$

with $\rho = 8$. From this representation, it is clear that K is a 0-symmetric polytope (it is even linearly parameterized by ρ, which would make a parametric derivation of memory allocations possible). To get an idea of the shape of K, consider an element of the form $A(b_r, b_c, 0, 0)$. It is written at time $64 \times (8 \times b_r + b_c)$ and it is read 8 iterations later, thus it conflicts with all elements of the next 8 written rows. An element of the form $A(b_r, b_c, 7, 0)$, written at time $64 \times (8 \times b_r + b_c)$, is read at the next iteration and thus conflicts only with the elements of the next written row. The next written row(s) can be written by S at the same iterations of the b_r and b_c loops (i.e., for $\delta_{br} = \delta_{bc} = 0$), but can also be written at the next iteration of the b_c loop (and same iteration of the b_c loop), or, extreme case (when $b_c = 63$), at the next iteration of the b_r loop and the very first iteration of the b_c loop (i.e., for $\delta_{br} = 1$ and $\delta_{bc} = -63$). The set K is 4-dimensional but, for clarity, it is better represented as the union of five 2-dimensional parts. The central part, which corresponds to the particular values $\delta_{br} = \delta_{bc} = 0$, is depicted in Figure 13. It is the square of all $(0, 0, \delta_r, \delta_c)$, where $-7 \leq \delta_r, \delta_c \leq 7$; it corresponds to a memory of 8 full rows, i.e., 64 elements. The set depicted in Figure 14 corresponds to two parts, the set of all (δ_r, δ_c) for $\delta_{br} = 0$ and $\delta_{bc} = 1$, and for $\delta_{br} = 1$, $\delta_{bc} = -63$. Its symmetric with respect to 0 is the set of all (δ_r, δ_c) for $\delta_{br} = 0$ and $\delta_{bc} = -1$, and for $\delta_{br} = -1$, $\delta_{bc} = 63$. These five pieces form the whole set K. To get yet a better view of the set K, consider again the set depicted in Figure 7. It is also a representation of the projection of K onto the first two components δ_{br} and δ_{bc}. In other words, for the two first indices, the situation is similar to Example 1.

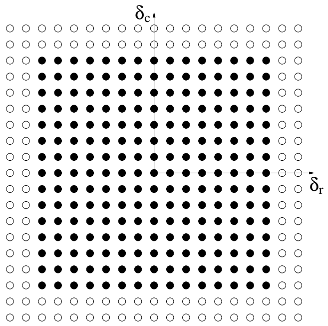

Figure 13: K part for $(\delta_{br}, \delta_{bc}) = (0, 0)$.

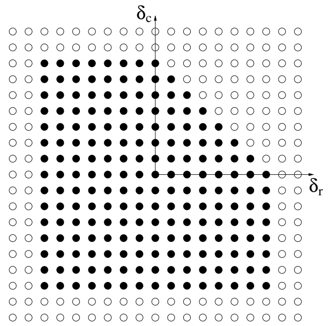

Figure 14: K part for $(\delta_{br}, \delta_{bc}) = (0, 1)$ and $(\delta_{br}, \delta_{bc}) = (1, -63)$.

We can now derive modular allocations. If we apply Heuristic 3 in the canonical basis, we find successively $\rho_{br} = 2$ and $\rho_{bc} = 2$ (as for Example 1), then we need to consider the set of Figure 13 and we get $\rho_r = 8$, and finally $\rho_c = 8$. This leads to the mapping $(b_r \bmod 2, b_c \bmod 2, r \bmod 8, c \bmod 8)$ with a memory size equal to 256. This is exactly what Lefebvre and Feautrier find since, here, we picked the basis given by the schedule, which is also the basis of the original code. If the original code is written with the b_c loop outside the b_r loop, but scheduled the same way with the b_r loop outside, then Lefebvre and Feautrier consider the index b_c first and they find successively $\rho_{bc} = 64$, $\rho_{br} = 1$, $\rho_r = 8$, and finally $\rho_c = 8$, for a memory size equal to 4096! Again, we see that the choice of basis, and the order in which we evaluate the F_i^*'s for Heuristic 3 (in the opposite order of the F_i's for Heuristic 2) is important.

Following our linearization mechanism for Heuristic 3, we can find a linear allocation with the same memory size than the mapping $(b_r \bmod 2, b_c \bmod 2, r \bmod 8, c \bmod 8)$ we just found; this is the mapping $b_r + 2b_c + 4r + 32c \bmod 256$.

As for Example 1, we also see that we can follow the schedule more closely, with the index $t = 64b_r + b_c$ (i.e., coalescing the two outer loops) so as to try to find a modular allocation with a memory size equal to 2 instead of 4 for the two outermost indices. If we redefine K with the indices (t, r, c) instead of (b_r, b_c, r, c), we get a 3-dimensional space with three 2-dimensional parts, the central part of Figure 13 for $t = 0$, the part of Figure 14 for $t = 1$, and its symmetric with respect to 0 for $t = -1$. Since K is now a 3-dimensional space instead of a 4-dimensional space, we are more likely to gain a factor 2 (as in the worse case of the heuristics). Indeed, we find successively $\rho_t = 2$, $\rho_r = 8$, and $\rho_c = 8$, and the mapping $(t \bmod 2, r \bmod 8, c \bmod 8)$, or equivalently $(b_c \bmod 2, r \bmod 8, c \bmod 8)$, with memory size 128, half of what we found before. The linearized version is $b_c + 2r + 16c \bmod 128$. Here, we may also consider the indices in the opposite order, with the same result, $\rho_c = 8$, $\rho_r = 8$, and $\rho_t = 2$. Then, the linearized version is $c + 8r + 64t \bmod 128$, which is a particular linearization of the original array, modulo 128, solution that De Greef, Catthoor, and De Man find in this particular case (again, by "luck" because the schedule is aligned with the array accesses).

Actually, the maximal distance between conflicting indices, in the linearized representation $c + 8r + 64b_c + 4096b_r$, is not 127 but 120, therefore De Greef, Catthoor, and De Man find the mapping $c + 8r + 64b_c + 4096b_r \bmod 121$, or equivalently $c + 8r + 64b_c + 103b_r \bmod 121$. Can we do better? To answer this question, we can search, as suggested in Section 4.3, for the smallest (in determinant) strictly admissible integer lattice for K, generating triangular matrices for the basis of the lattices we try. The numbers involved here are small enough to make this search practical. We find that the best modular allocation has a memory size equal to 112, for the (unique) lattice generated by the vectors $(1, 0, 0, 12), (0, 1, 0, 12), (0, 1, 4, 20), (0, 0, 0, 28)$, which corresponds (among all allocations with same associated lattice) to the allocation $(r \bmod 4, 16(b_r + b_c) + 2r + c \bmod 28)$, a two-dimensional allocation (with no corresponding linearized version). (Note that $16t = 16(64b_r + b_c) = 16(b_r + b_c) \bmod 28$.) The best one-dimensional allocation needs one more memory cell, with memory size 113, for example, the allocation $60b_r + 8b_c + 42r + c \bmod 113$, i.e., $8t + 42r + c \bmod 113$, or the more "natural" allocation $64t + 8r + 3c \bmod 113$.

8 Conclusion

We have proposed herein a mathematical framework to study modular memory allocations. We have shown a fundamental link between valid modular allocations and strictly admissible lattices for the set $\mathcal{D}$ of differences of conflicting indices, as well as the equality of the memory used by the former and the determinant of the latter. We have given upper and lower bounds connecting the memory size achievable by a modular mapping and the volume of $\mathcal{D}$, in the case where this set is, or is approximated by, a 0-symmetric polytope. We explored several heuristic mechanisms that allow us to derive optimal modular allocations, up to a multiplicative factor that depends on the problem dimension only. Several important questions remain open, both from theoretical and practical viewpoints.

With our assumptions, we showed that *one-dimensional* mappings (which are important in practice) are optimal up to a multiplicative factor (Corollary 1). We also showed how to build a mapping with minimal number of dimensions among equivalent mappings (Theorem 1). One question remains concerning one-dimensional mappings: Exactly how much do we lose when restricting to such mappings as opposed to multi-dimensional mappings?

Can we better exploit the fact that $\mathcal{D}$ is not an arbitrary 0-symmetric convex body K, but rather *comes from a scheduled program*? For this, we proposed the strategy, often quite good in the case of a sequential multi-dimensional affine schedule coupled to a mapping expressed in terms of loop indices, wherein we blend the techniques of Lefebvre–Feautrier with those of Quilleré–Rojopadhye; that is, we build Lefebvre–Feautrier moduli, but using the basis defined by the schedule as in Quilleré–Rajopadhye. Can we identify classes of programs, in a more formal way, for which we can be sure that a simple-to-compute choice of basis will give good performance?

Can we derive *better heuristics* to approximate $\Delta_Z(K)$, in theory (i.e., improve the bound $n!\mathrm{Vol}(K)$), and in practice? In particular, maybe there are better heuristics based on the construction of a strictly admissible (possibly *rational*) lattice that can be converted into a strictly admissible *integer* lattice?

In which cases can modular allocations be arbitrarily bad compared to MAXLIVE? In general, how bad can optimal modular allocations be compared to MAXLIVE, i.e., what do we lose with modular allocations compared to *general allocations*? Of course, if the program is completely irregular, using modular allocations is obviously sub-optimal. But what can we say for an affine program for example, with reasonable assumptions?

Another topic worthy of study is the effect of the approximations used to build $\mathcal{D}$, and the connection of those approximations to the representation of dependences and schedules. Can we generalize our heuristics and our theories to handle representation of $\mathcal{D}$ by a union of polytopes rather than a single polytope?

This paper takes a first step towards addressing these questions: we have focused on the mathematical framework and its general properties. Future work should take up some more sharply reasoned theoretical questions as well as effective practical implementation.

Dedication

This problem came to us as part of the PICO research effort at HP Labs, led by Bob Rau. Bob and the other members of the team had a hand in launching our work on it.

Despite his many and varied duties, Bob remained a keen and gifted engineer and scientist right up to the end of his career. We were lucky enough to have had over twenty years to work with and know him. It wasn't nearly enough. We fondly dedicate this work to his memory.

References

[1] Shail Aditya and Michael S. Schlansker. Shiftq: A buffered interconnect for custom loop accelerators. In *International Conference on Compilers, Architecture, and Synthesis for Embedded*

Systems (CASES'01), pages 158–167, Atlanta, USA, 2001. ACM Press.

[2] Paul Budnik and David J. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, C-20:1566–1569, December 1971.

[3] Francky Catthoor et al. Atomium: A toolbox for optimising memory I/O using geometrical models. http://www.imec.be/design/atomium/.

[4] Alain Darte. Mathematical tools for loop transformations: From systems of uniform recurrence equations to the polytope model. In M. H. Heath, A. Ranade, and R. S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105 of *IMA Volumes in Mathematics and its Applications*, pages 147–183. Springer Verlag, 1998.

[5] Alain Darte, Michèle Dion, and Yves Robert. A characterization of one-to-one modular mappings. *Parallel Processing Letters*, 5(1):145–157, 1996.

[6] Alain Darte, Rob Schreiber, Bob Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):159–172, 2002.

[7] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In 6*th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*, October 2003.

[8] Alain Darte, Frédéric Vivien, and Yves Robert. *Scheduling and Automatic Parallelization*. Birkhäuser, Boston, 2000.

[9] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[10] Cynthia Dwork. Lattices and their application to cryptography. Available as lecture notes from Stanford University, http://theory.stanford.edu/~csilvers/cs359/, 1998.

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

[12] Paul Feautrier. Automatic parallelization in the polytope model. In Alain Darte and Guy-René Perrin, editors, *The Data-Parallel Programming Model*, volume 1132 of *LNCS Tutorial*, pages 79–103. Springer Verlag, 1996.

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

[14] Peter M. Gruber. Geometry of numbers. In P.M. Gruber and J.M. Wills, editors, *Handbook of Convex Geometry*, volume B, chapter 3.1, pages 739–763. Elsevier Science Publishers B.V., 1993.

[15] Lou Hafer. The generalized basis reduction algorithm of Lovász and Scarf (annotated). http://www.cs.sfu.ca/~lou/MITACS/grb.pdf, June 2000.

[16] William Jalby, Jean-Marc Frailong, and Jacques Lenfant. Diamond schemes: An organization of parallel memories for efficient array processing. Technical Report 342, INRIA, Centre de Rocquencourt, 1984.

[17] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, September 2002.

[18] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'00)*, San Diego, May 2000.

[19] Jeffrey C. Lagarias. Point lattices. In R. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume I, chapter 19, pages 919–966. Elsevier Science Publishers B.V., 1995.

[20] Vincent Lefebvre and Paul Feautrier. Storage management in parallel programs. In *Fifth Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, London, UK, January 1997. IEEE Computer Society Press.

[21] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[22] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[23] László Lovász. *An Algorithmic Theory of Numbers, Graphs, and Convexity*, volume 50 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, 1986.

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

[25] Morris Newman. *Integral Matrices*. Academic Press, 1972.

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

[27] Patrice Quinton et al. Alpha homepage: A language dedicated to the synthesis of regular architectures. http://www.irisa.fr/cosi/ALPHA.

[28] Henry D. Shapiro. Theoretical limitations on the efficient use of parallel memories. *IEEE Transactions on Computers*, C-27(5):421–428, May 1978.

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

[30] Synfora. http://www.synfora.com.

[31] G. Tel, Jan Van Leeuwen, and Harry A. G. Wijshoff. The one dimensional skewing problem. Technical Report RUU-CS-89-23, Rijksuniversiteit Utrecht, the Netherlands, October 1989.

[32] G. Tel and Harry A. G. Wijshoff. Hierarchical parallel memory-systems and multi-periodic skewing schemes. Technical Report RUU-CS-85-24, Rijksuniversiteit Utrecht, the Netherlands, August 1985.

[33] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman 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.

[34] Remko Tronçon, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Storage size reduction by in-place mapping of arrays. In A. Cortesi, editor, *Verification, Model Checking and Abstract Interpretation, Third Int. Workshop, VMCAI 2002,*, volume 2294 of *LNCS*, pages 167–181. Springer Verlag, 2002.

[35] Alexandru Turjan and Bart Kienhuis. Storage management in process networks using the lexicographically maximal preimage. In 14*th International Conference on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, June 2003. IEEE Computer Society Press.

[36] Harry A. G. Wijshoff and Jan Van Leeuwen. Periodic versus arbitrary tessellations of the plane using polyominos of a single type. Technical Report RUU-CS-82-11, Rijksuniversiteit Utrecht, the Netherlands, July 1982.

[37] Harry A. G. Wijshoff and Jan Van Leeuwen. Periodic storage schemes with a minimum number of memory banks. Technical Report RUU-CS-83-4, Rijksuniversiteit Utrecht, the Netherlands, February 1983.

[38] Harry A. G. Wijshoff and Jan Van Leeuwen. The structure of periodic storage schemes for parallel memories. *IEEE Transactions on Computers*, C-34(6):501–505, June 1985.