

Lattice-Based Memory Allocation

Alain Darté, Robert Schreiber, and Gilles Villard

A. Darté and G. Villard: CNRS, LIP, École normale supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France, Alain.Darte@ens-lyon.fr, Gilles.Villard@ens-lyon.fr

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

Abstract

We investigate the problem of memory reuse in order to reduce the memory needed to store an array variable. We develop techniques that can lead to smaller memory requirements in the synthesis of dedicated processors or to more effective use by compiled code of software-controlled scratchpad memory.

Memory reuse is well-understood for allocating registers to hold scalar variables. Its extension to arrays has been studied recently for multimedia applications, for loop parallelization, and for circuit synthesis from recurrence equations. In all such studies, the introduction of modulo operations to an otherwise affine mapping (of loop or array indices to memory locations) achieves the desired reuse. We develop here a new mathematical framework, based on critical lattices, that subsumes the previous approaches and provides new insight.

We first consider the set of indices that conflict, those that cannot be mapped to the same memory cell. Next we construct the set of differences of conflicting indices. We establish a correspondence between a valid modular mapping and a strictly admissible integer lattice – one having no nonzero element in common with the set of conflicting index differences. The memory required by an optimal modular mapping is equal to the determinant of the corresponding lattice. The memory reuse problem is thus reduced to the (still interesting and nontrivial) problem of finding a strictly admissible integer lattice of least determinant. We then propose and analyze several practical strategies for finding strictly admissible integer lattices, either optimal or optimal up to a multiplicative factor, and hence memory-saving modular mappings. We explain and analyze previous approaches in terms of our new framework.

Index Terms

Program transformation, memory size reduction, admissible lattice, successive minima.

I. INTRODUCTION

We propose a mathematical framework and heuristics for solving the problem of mapping from array or loop indices of a given scheduled program, to a set of memory locations as small as possible for the intermediate and final results of the program. The mapping *reuses memory locations* by storing several elements of an array in one location. To do so does not violate program semantics if the elements that share a memory location never simultaneous hold live values under the given schedule. We give a lattice-based mathematical framework that subsumes previous approaches and gives new insights, and we apply our model for finding guaranteed space-saving mappings.

We consider the class of **linear storage allocations** and represent the constraints

imposed by the program semantics in terms of a set \mathcal{D} of differences of **conflicting indices** (Section II). Our results are based on a correspondence between valid linear allocations and integer lattices Λ such that $\mathcal{D} \cap \Lambda = \{\vec{0}\}$ (**strictly admissible** lattices for \mathcal{D}). We study a natural equivalence relation on linear allocations: two allocations are equivalent if they are equal up to a renaming of the memory locations they use. Our first result is to establish that, in each equivalence class, there is a **modular mapping** (an allocation, into a multidimensional array, expressed as an affine function, followed by modulo operations) that requires the smallest (among all allocations of this class) array size and array dimension (Section III, Thm. 1). Then, through lattice theory and geometric reasoning, this equivalence leads us to new lower and upper bounds, and to an optimal allocation construction (Section III-D). The third aspect of our contribution is new **guaranteed approximation heuristics** that unify most previously existing ones [1]–[3], and their analysis (Section IV).

For illustration, we will apply our results on a detailed case study (Section VII): the code in Fig. 1 accesses a 4D array $A(b_r, b_c, r, c)$, in two pipelined communicating loops that write every “rows” $A(b_r, b_c, r, *)$, and read every “columns” $A(b_r, b_c, *, c)$. For making things simpler concerning the *scheduling function* (see Section II-B), we assume that each operation of S writes all elements of a row in “parallel”, *i.e.*, at the same “macro-time” $(64 \times b_r + b_c) \times 8 + r$ (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 $\rho = 8$ is such that dependences are respected. On this example, our problem is to find a linear

DO $b_r = 0, 63$	DO $b_r = 0, 63$
DO $b_c = 0, 63$	DO $b_c = 0, 63$
DO $r = 0, 7$	DO $c = 0, 7$
S: $A(b_r, b_c, r, *) = \dots$	T: $\dots = A(b_r, b_c, *, c)$
ENDDO	ENDDO
ENDDO	ENDDO
ENDDO	ENDDO

Fig. 1. DCT-like example, with two pipelined loops.

allocation to an intermediate buffer, as small as possible, for storing the entries of A between corresponding write and read instructions. A typical modular mapping solution here is given by $(b_r \bmod 2, b_c \bmod 2, r \bmod 8, c \bmod 8)$, *i.e.*, by a temporary storage in

a 4D buffer of size 256 (obtained using [3]). We will see that various other solutions can be built, and apply our heuristics for constructing 1D mappings (*i.e.*, implemented with only one mod) such as $b_r + 2b_c + 4r + 32c \bmod 256$, or an optimal solution ($r \bmod 4, 16(b_r + b_c) + 2r + c \bmod 28$) of size 112. For this optimal size, there are two equivalence classes (symmetry in c and r), leading to 2D solutions, with no equivalent 1D solution.

Our methods can be applied to automatic hardware synthesis ¹ for reducing the size of buffers. They can also be used by compilers to make efficient use of limited software-managed scratchpad storage. Several heuristics have been proposed to reduce the memory needed by a scheduled program (see Section V), but (as stated in [12]) until now there has been no satisfactory theory that relates the interplay of scheduling (*i.e.*, *when* statements can be executed) and storage (*i.e.*, *where* their results can be stored). De Greef *et al.* [1] propose to work with the array indices, choose one canonical linearization of the array (*i.e.*, one permutation of its axes) then follow this by a modulo operation that wraps the set of “virtual” memory locations into a smaller set of actual memory locations. Lefebvre and Feautrier [3] propose to work with the loop rather than array indices, and to wrap with a modulo operation applied to each loop index. For example, the result of statement S in iteration (i, j) is stored in a dedicated array A_S at location $A_S(i \bmod b_i, j \bmod b_j)$, where b_i and b_j are chosen to guarantee correctness and minimize storage. Quilleré and Rajopadhye [2] also use a dedicated array A_S , but the loop indices first undergo an affine mapping before a modulo operation is applied to one of the resulting array indices. More recently, Thies *et al.* [12], extending the work of Strout *et al.* [13], consider wrapping with a modulo operation in a single dimension and how to find a good one. A novel aspect is that they obtain memory reductions valid for *any* schedule that respects the dependences of the program.

All these techniques use mappings that are special cases of the modular mappings we study in this paper: storage allocations that access multidimensional arrays with an affine function followed by a modulo operation in each dimension. Thies *et al.*, aware of the limitations of their technique, 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*”

¹See, e.g., the HP Labs PICO project [4], [5] (now a product of a new company, Synfora [6]), the IMEC Atomium project [1], [7], [8], the Compaan project [9], [10], and the Alpha project [2], [11].

vector and having an efficient implementation". This is exactly our goal here; to develop a mathematical framework that allows us to capture all suitable, space-saving storage allocations, to define optimality, to propose heuristics, and to discuss the quality of (ours and others) heuristics and measure them by this standard (see Sections V and VI).

The paper is organized as follows. In Section II, we model the problem of memory reuse in a scheduled program. We focus on linear allocations and define their validity via a set of conflicting indices. Section III is concerned by the correspondence between valid allocations and strictly admissible lattices, including an optimal construction. Most of these results are obtained under the very practical assumption that the set of differences of conflicting indices is represented or approximated as the set of integer points in a polytope K . We derive several bounds and show, especially, that any valid modular mapping needs at least $\text{Vol}(K)/2^n$ locations. In Section IV, we introduce heuristics that use less than $c_n \text{Vol}(K)$ locations (c_n depends on the dimension n but not on K); hence they are optimal up to a multiplicative factor. These heuristics rely either on arrays of smallest dimension (Thm. 1), or on 1D mappings (Corollary 1). After a description of previous approaches in Section V, we relate them to our results with a discussion in Section VI. Section VII is the announced detailed case study, before concluding in the final section.

NOTA. The reader may refer to [14] for a more comprehensive document, including all proofs, and more detailed examples and discussions.

II. INDICES AND MEMORY ALLOCATION

Here we introduce the array memory reuse problem. We define the basic object we need to build from the scheduled program, the set \mathcal{C} of pairs of conflicting indices corresponding to data that may not share the same memory location. Then, we introduce a general class of space-saving storage allocation mappings.

A. The set of conflicting indices

We are interested in finding a storage-saving mapping $\sigma_A(\vec{i})$ from the index space of a single array A to a set of storage locations indexed by addresses.

Some authors ignore the arrays of the given program, and take the view that what must be stored and communicated (from definitions to uses) are the values created by program execution. Every value is represented by the statement S that produced it and the

iteration vector \vec{i} of the surrounding loops. Thus, this transformation can be viewed and implemented by the use of a big virtual array dedicated to the statement S , in which each element is written once. They then seek to reduce the memory required by choosing an allocation map σ_S from iteration vectors to array indices in a (smaller) multidimensional array A_S . The elements of A_S are mapped to memory locations without further reuse. This appears to allow mappings of values to memory that are not constrained by the programmer's choice of loop index to array index map, but rather can be whatever is best for the given program. To use it, however, all right-hand side array references must be replaced with references to A_S in such a way that the correct element is accessed. And to do this requires *exact* dependence analysis, which is not generally feasible.

We can sidestep this question; our approach is equally applicable whether or not we use single-assignment code. The program we consider may be the original program, or it may have undergone preliminary program transformations and optimizations, including the change of some arrays and statements to single-assignment form as above. In either case, the transformed program makes array references, the arrays must be stored in memory, and if we are given the program schedule we can take advantage of memory reuse to minimize the actual storage requirement. Single assignment code should offer us ample opportunities to find such reuse, but it does not change anything about our techniques.

Thus, without loss of generality, for us data are identified by array indices.

B. Schedules and conflicting array indices

Next, we assume given a scheduling function θ which assigns to each operation u a "virtual" execution time, *i.e.*, an element of a totally ordered set (\mathcal{T}, \preceq) . The schedule may express parallelism – some operations are simultaneously scheduled. The operation u will be computed strictly before the operation v iff (if and only if) $\theta(u) \prec \theta(v)$. A valid schedule respects all dependences: in particular, 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 , or \mathbb{Z}^d with the lexicographic order (for a **multidimensional schedule**), and the function θ , when restricted to operations $u = (S, \vec{i})$, is an affine function of \vec{i} . But these restrictions are not essential: our theoretical results apply no matter what the schedule.

We need to characterize the indices corresponding to array elements that can be mapped to the same memory location.

Definition 1 (Conflicting indices): The array indices \vec{i} and \vec{j} conflict (denoted by $\vec{i} \bowtie \vec{j}$)

if the corresponding array elements are simultaneously live under the schedule θ . (By convention we say too that $\vec{i} \bowtie \vec{i}$.)

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 a multidimensional generalization of the interference graph for register allocation. The well-known quantity MAXLIVE (the largest number of values simultaneously live) is equal to the size of the largest set S such that $S \times S \subseteq CS$; any such S is a set of mutually conflicting indices. Define $DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$, the **conflicting index difference set**. Because \bowtie is symmetric, DS is 0-symmetric ($x \in DS$ if and only if $-x \in DS$); because \bowtie is reflexive, $0 \in DS$.

Let us see how we can define the relation \bowtie exactly in the special case in which the operation (S, \vec{i}) writes to location $A_S(\vec{i})$. 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. For each iteration vector $\vec{i} \in \overline{\mathcal{I}}_S$, let $L(S, \vec{i})$ be the last operation that reads the value computed by the operation (S, \vec{i}) . Then, the relation \bowtie is defined by:

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

In the simple case of a 1D schedule, $\vec{i} \bowtie \vec{j}$ when the intervals $[\theta(S, \vec{i}), \theta(L(S, \vec{i}))]$ and $[\theta(S, \vec{j}), \theta(L(S, \vec{j}))]$ overlap.

As a practical matter, we recommend using polyhedra (or a union of polyhedra) to represent the point sets that arise. In the common case, when the schedule θ is piecewise affine, when $\overline{\mathcal{I}}_S$ can be described as all integer points in a polytope, and when all accesses to arrays are affine, then $\vec{i} \mapsto L(S, \vec{i})$ is piecewise affine (see [2], [3]) and can be obtained with standard techniques to compute lexicographic minima or maxima in polytopes. Furthermore, CS can be represented as all integer points in a finite union of polytopes (when $L(S, \vec{i})$ is piecewise affine) or simply a polytope (when $L(S, \vec{i})$ is affine). As noted in [2], it may be interesting to first decompose $\overline{\mathcal{I}}_S$ into disjoint subdomains (each operation (S, \vec{i}) will then write in 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 as we show in Section IV, storage allocation heuristics can be shown to perform well for polytopes.

When exact analysis is not possible (or not desired), we can fall back on an approxi-

mation \mathcal{C} to CS , as long as it is a super-approximation, *i.e.*, $CS \subseteq \mathcal{C}$. For example, when reasoning with array indices, instead of considering exact lifetimes, we can say that an array element is live from the first time it is written to the last time it is read (even if it is dead for some parts of this period, and is written again). With such a definition, there is no need to precompute any quantity similar to $L(S, \vec{i})$. Indeed, suppose, to simplify, that the program has only two statements, S and T , that, for each $\vec{i} \in \mathcal{I}_S$, (S, \vec{i}) writes to $A(f(\vec{i}))$, and that, for each $\vec{k} \in \mathcal{I}_T$, (T, \vec{k}) reads from $A(g(\vec{k}))$ a value computed by S . Then, even if f and g are not one-to-one, we can define an approximation \mathcal{C} for CS as $\{(\vec{a}, \vec{c}) \mid \vec{i}, \vec{j} \in \mathcal{I}_S, \vec{k}, \vec{l} \in \mathcal{I}_T, \vec{a} = f(\vec{i}) = g(\vec{k}), \vec{c} = f(\vec{j}) = g(\vec{l}), \theta(S, \vec{i}) \preceq \theta(T, \vec{k}), \theta(S, \vec{j}) \preceq \theta(T, \vec{l}), \theta(S, \vec{i}) \preceq \theta(T, \vec{l}), \theta(S, \vec{j}) \preceq \theta(T, \vec{k})\}$, *i.e.*, $A(\vec{a})$ and $A(\vec{c})$ are both written, and both read later, so they should not share the same location.

To conclude this discussion, 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 case, in which \mathcal{D} is equal to the set denoted $\overset{\circ}{K}$ of all integer points within a 0-symmetric polytope K .

C. Memory allocation

Given a scheduling function θ , the array memory allocation problem is to determine an allocation function that maps the elements of an array to a set of memory locations that is as small as possible. We first define linear allocations, which are the allocation functions we consider. We show them to be equivalent to **modular mappings**, which have a simple and usable form. We show that the correctness of a linear allocation can be determined by its behavior on the conflicting index difference set \mathcal{D} : in fact, its correctness can be determined simply by considering the intersection of \mathcal{D} and its kernel. Then, we give some general lower bounds on the storage required by a valid linear allocation.

1) *Allocation functions*: We seek an allocation function σ that specifies for each array element index (a n -dimensional integer vector), where the corresponding values will be stored in some dedicated, hopefully quite small, array of dimension p . We will get there eventually, when we discuss modular mappings. But we start with some more general mappings that turn out to be, in fact, not really more general. Note that n now denotes the dimension of the program array whose elements we are mapping to memory.

Standard array index to memory mappings map large arrays to large amounts of memory. On the contrary, we consider mappings whose target space is small:

Definition 2: An **allocation** (or **mapping**) σ of **size** $\text{size}(\sigma) = m$ is a function $\sigma : \mathbb{Z}^n \rightarrow \mathcal{M}$ where $\mathcal{M} \subset \mathbb{Z}^p$ is a finite set of m elements.

The allocations $\sigma : \mathbb{Z}^n \rightarrow \mathcal{M}_\sigma$ and $\tau : \mathbb{Z}^n \rightarrow \mathcal{M}_\tau$ are **equivalent** if $\tau = \phi \circ \sigma$ for some bijection $\phi : \mathcal{M}_\sigma \rightarrow \mathcal{M}_\tau$. Equivalent allocations are identical up to a renaming of the set of elements they map to, hence equivalent *surjective* allocations have the same size.

As a practical matter, we think it necessary to restrict the form of the allocation function as follows.

Definition 3: A **linear allocation** σ of **size** m is a morphism $\sigma : \mathbb{Z}^n \rightarrow (\mathcal{M}, \oplus)$ where (\mathcal{M}, \oplus) is a finite \mathbb{Z} -module (*i.e.*, finite Abelian group) of m elements.

The **kernel** of a morphism σ is $\ker(\sigma) = \{\vec{i} \mid \sigma(\vec{i}) = 0\}$ where 0 is the identity element of \mathcal{M} . With elementary morphism theory, it can be shown that two linear allocations are equivalent if and only if they have the same kernel (see [14]). For example, $i \bmod 2$ has the kernel $2\mathbb{Z}^n$, as does the equivalent linear allocation $2i \bmod 4$. The latter, viewed as a mapping into the group $\mathbb{Z}/4\mathbb{Z} = \{0, 1, 2, 3\}$ is not surjective, which is why its size (4) is not the same as the size of the former (2).

We now introduce the mappings we use in practical storage optimization, and that were used by the previous approaches (see Section V), the modular mappings.

Definition 4: A **modular mapping** (M, \vec{b}) , defined by a $p \times n$ integer matrix M and a positive integer vector \vec{b} of dimension p , maps the index \vec{i} to $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (the modulo operation is applied componentwise) in a p -dimensional array of shape \vec{b} .

A modular mapping may reduce the dimensionality of the dataset; this occurs when M has fewer rows than columns. Note too that when some components of \vec{b} are equal to 1, we can forget about them and the corresponding row(s) of M , and reduce p .

A modular mapping reuses memory in two ways. First, if $\vec{i} - \vec{j}$ is in the null space of M , then the corresponding array elements are mapped identically. Second, if $M(\vec{i} - \vec{j}) \equiv \vec{0} \bmod \vec{b}$, then the corresponding array elements are likewise mapped to the same memory location. Technically, however, the first condition is a special case of the second.

A modular mapping is a linear allocation. The target set is the finite rectangle $\mathcal{M} = \mathbb{Z}/b_1\mathbb{Z} \times \mathbb{Z}/b_2\mathbb{Z} \times \cdots \times \mathbb{Z}/b_p\mathbb{Z}$ and its size is the product of the elements of \vec{b} : $m = \prod_i b_i$. If we use a modular mapping, this is how much storage we need. The number of elements of the target actually used may be smaller than this, in other words, a modular mapping may or may not be surjective. We shall show in the next section that every linear allocation is equivalent to a family of modular mappings, at least one of which is a surjection.

2) *C-valid allocations*: We now characterize valid allocations, *i.e.*, those that preserve the program semantics. Following Section II-A, we assume that the constraints to respect are specified by a set $\mathcal{C} \subset \mathbb{Z}^n \times \mathbb{Z}^n$ of pairs of potentially conflicting indices. The mapping σ is **C-valid** if $(\vec{i}, \vec{j}) \in \mathcal{C}$, $\vec{i} \neq \vec{j} \Rightarrow \sigma(\vec{i}) \neq \sigma(\vec{j})$. It is immediate that if σ and τ are equivalent allocations, σ is \mathcal{C} -valid iff τ is \mathcal{C} -valid. For *linear* allocations, this validity requirement can be expressed in terms of \mathcal{D} rather than \mathcal{C} . A linear allocation σ is \mathcal{C} -valid iff $\vec{d} \in \mathcal{D}$, $\vec{d} \neq \vec{0} \Rightarrow \sigma(d) \neq \vec{0}$. We thus have the following characterization.

Proposition 1: A linear allocation is \mathcal{C} -valid iff $\mathcal{D} \cap \ker \sigma = \{\vec{0}\}$.

From these definitions, we can derive some lower bounds on the storage size achievable by a \mathcal{C} -valid allocation and by a \mathcal{C} -valid linear allocation: $\min\{\text{size}(\sigma) \mid \sigma \text{ is } \mathcal{C}\text{-valid}\} \geq \text{MAXLIVE} = \max\{\text{Card}(S) \mid S \times S \subseteq \mathcal{C}\}$ ($\text{Card}(S)$ is the number of integer points in S) since all points in such a set S are simultaneously live and must be mapped to different locations. If σ is a *linear* \mathcal{C} -valid allocation and there is a set S such that $S - S \subseteq \mathcal{D}$, then σ is also $(S \times S)$ -valid hence:

$$\min\{\text{size}(\sigma) \mid \sigma \text{ is } \mathcal{C}\text{-valid}\} \geq \max\{\text{Card}(S) \mid S - S \subseteq \mathcal{D}\}. \quad (1)$$

Example 1: Consider the code fragment in Fig. 2 and suppose that we may reuse the memory for A , *i.e.*, A is not “live-out” from the code fragment. Suppose that the two loops

DO $i = 0, N - 1$	DO $i = 0, N - 1$
DO $j = 0, N - 1$	DO $j = 0, N - 1$
S: $A(i, j) = \dots$	T: $B(i, j) = A(i, j) + \dots$
ENDDO	ENDDO
ENDDO	ENDDO

Fig. 2. Code for Example 1, the second loop starts 1 clock cycle after the first.

are scheduled sequentially as written, but that the second loop is pipelined, with respect to the first one, one clock cycle later. In other words, for all i, j , $\theta(S, (i, j)) = (i, j)$, and $\theta(T, (i, j)) = (i + 1, 0)$ if $j = N - 1$, $\theta(T, (i, j)) = (i, j + 1)$ otherwise. We can also use a 1D 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 parallel processors or hardware blocks, the values of A being placed in an intermediate buffer whose access function and size are to be designed.

If we consider that a value is dead only at the end of the read in T , then each array element is live during 2 cycles and only 2 values are live at the same time. We therefore only need 2 memory cells or registers to store them. The mapping $A(i, j) \mapsto Ni + j \bmod 2$ (plus a base address in memory) is a valid mapping that requires 2 memory cells, but how can a compiler automatically find such a mapping?

In Fig. 3, we pictorially represent, for some particular index vector \vec{i} (in black), all other index points \vec{j} (in grey) that correspond to operations (S, \vec{j}) executed *after* the operation (S, \vec{i}) , but *before* the time at which the value written by (S, \vec{i}) is read, *i.e.*, the iterations for which $\vec{i} \bowtie \vec{j}$. For (i, j) with $j < N - 1$, $S(i, j + 1)$ should not write in the

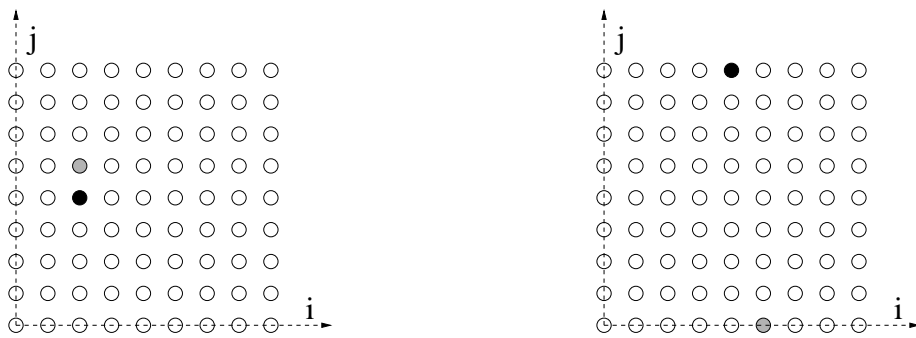
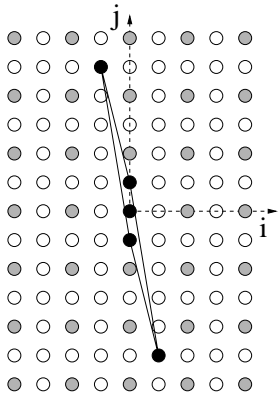
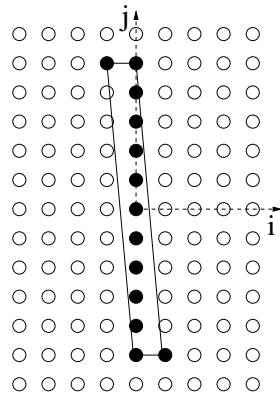


Fig. 3. Two different cases of conflicting iterations for the code of Fig. 2, for $N = 9$.

same location as $S(i, j)$ (case on the left) and $S(i + 1, 0)$ should not write in the same location as $S(i, N - 1)$ (case on the right). In this example, the iteration space for S and the array space for A are perfectly aligned (statement S at iteration (i, j) writes in A at location (i, j)), therefore Fig. 3 can be interpreted in terms of array locations: for $A(i, j)$ (in black), the other points (in grey) represent the array elements that should not be mapped to the same memory address as $A(i, j)$.

We see that there are only 5 different vectors in DS , $(0, 0)$, $(0, 1)$, $(1, 1 - N)$, and their negations (see the set DS in Fig. 4 as well as a polytope K such that $DS = \overset{\circ}{K}$). In grey, some regularly spaced points are represented; these are the elements in the kernel of the modular mapping $(i, j) \mapsto (i \bmod 2, j \bmod 2)$, whose intersection with DS is equal to $\{\vec{0}\}$, and which is thus valid. In Fig. 5, we represent a super-approximation \mathcal{D} to DS . Now, the mapping $(i, j) \mapsto (i \bmod 2, j \bmod 2)$ is no longer valid. Actually, since $S = \{(0, 0), (0, 1), \dots, (0, N - 1)\}$ is such that $S - S \subseteq \mathcal{D}$, any valid linear allocation requires at least N memory cells. Thus, with this apparently slight approximation, we lose an order of magnitude (when N is large) in memory size. \square


 Fig. 4. The exact set DS for Example 1.

 Fig. 5. An approximation $\mathcal{D} = \overset{\circ}{K}$ of the set DS .

III. INTEGER LATTICES AND LINEAR ALLOCATIONS

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 a **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 to us since, as we will see, they are in correspondence with equivalence classes of 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} .

An integer matrix U is **unimodular** if its determinant has absolute value one, or alternatively if it has an integer inverse. Unimodular matrices are isomorphisms of \mathbb{Z}^n : if U is unimodular, then $\mathbb{Z}^n = U\mathbb{Z}^n$. Given a matrix M of rank n in $\mathbb{Z}^{n \times n}$, there exist a unimodular matrix U and an upper triangular matrix H such that $H = MU$; moreover H can be chosen such that $0 \leq h_{i,j} < h_{i,i}$ for all $j > i$. The matrix H is called the **Hermite** normal form [15] of M , it is not changed if M is multiplied on the right by a unimodular matrix. Given $M \in \mathbb{Z}^{n \times n}$ of rank n , there exist two unimodular matrices U_1 and U_2 and an integer diagonal matrix $S = \text{diag}(s_1, \dots, s_n)$ such that $S = U_1 M U_2$ and s_i divides s_{i+1} , for $1 \leq i < n$. The diagonal factor S is called the **Smith** form of M [15], it is not changed if M is multiplied on either side by a unimodular matrix.

A. Kernels and the representation of linear allocations

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 ; for otherwise the image of \mathbb{Z}^n under σ would be infinite, and we know it is not.

Proposition 2: Let $\Lambda \subseteq \mathbb{Z}^n$ be a lattice of rank n . One can compute a modular mapping (U, \vec{s}) with kernel Λ , with U unimodular in $\mathbb{Z}^{n \times n}$, \vec{s} such that s_i divides s_{i+1} , and $d(\Lambda) = \prod_{i=1}^n s_i$. Moreover, any linear allocation is equivalent to such a modular mapping.

Proof: Consider a matrix $A \in \mathbb{Z}^{n \times n}$ whose columns are a basis of Λ : $\Lambda = AZ^n$. Write $S = U_1AU_2$, the Smith form of A , and let $U = U_1$ and \vec{s} such that $S = \text{diag}(\vec{s})$. Then $U\vec{x} \bmod \vec{s} = 0$ 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 $\vec{x} = U_1^{-1}SU_2^{-1}\vec{v} = A\vec{v}$. 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 $\Lambda = \ker(\sigma)$ for a linear allocation σ . Then (U, \vec{s}) has kernel Λ and is thus equivalent to σ . (NOTA. For a modular mapping σ , $\ker(\sigma)$ can be computed with integer matrix computations, as in [16].) ■

As already mentioned, for a modular mapping (M, \vec{b}) , the rows of M corresponding to $b_i = 1$ may be omitted since the corresponding dimension of the storage array is not used. Proposition 2 thus shows that the study of linear allocations $\sigma : \mathbb{Z}^n \rightarrow \mathcal{M}$ reduces to the study of (surjective) modular mappings $\vec{i} \mapsto U\vec{i} \bmod \vec{s}$ where $U \in \mathbb{Z}^{p \times n}$ can be completed into a unimodular matrix and where \vec{s} has no unit entries. For a linear allocation σ , the construction of Proposition 2, with $\Lambda = \ker(\sigma)$, computes an equivalent modular mapping of the form (U_p, \vec{s}_p) , where $U_p \in \mathbb{Z}^{p \times n}$ is formed by the last p rows of U and \vec{s}_p is given by the nonunit entries of \vec{s} . The following theorem shows that this strategy leads to a modular mapping that uses the *fewest dimensions* and the least storage.

Theorem 1: Let σ be a linear allocation. The modular mapping (U_p, \vec{s}_p) given by the construction of Proposition 2, with $\Lambda = \ker(\sigma)$, requires the smallest storage size $d(\Lambda)$ among all linear allocations equivalent to σ and the fewest array dimensions p among all modular mappings equivalent to σ .

Proof: Let (U, \vec{s}) be the modular mapping of Proposition 2 built from the kernel Λ of σ . By construction, $U^{-1}S$, with $S = \text{diag}(\vec{s})$, is a basis of Λ . The size of (U, \vec{s}) is $d(\Lambda)$ and, as U is unimodular, it is surjective. Elementary group theory shows that the size of a linear allocation σ' , equivalent to σ (*i.e.*, with underlying lattice Λ), defined from \mathbb{Z}^n to

an abelian group \mathcal{M} , is a multiple of $d(\Lambda)$ as $d(\Lambda)$ is the number of elements in $\sigma'(\mathbb{Z}^n)$, which is a subgroup of \mathcal{M} . Therefore, (U, \vec{s}) requires the smallest storage size.

Let (M, \vec{b}) be a modular mapping equivalent to σ . We first show that $B = \text{diag}(\vec{b})$ and $S = \text{diag}(\vec{s})$ are such that $B = PSQ$ for integer matrices P and Q in $\mathbb{Z}^{n \times n}$. Write $S_{\vec{b}} = \text{diag}(\vec{b}_S) = U_1 B U_2$, 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 $M\vec{x} = B U_2 \vec{u}$ iff there exists $\vec{v} \in \mathbb{Z}^n$ (with $\vec{v} = U_2 \vec{u}$) such that $M\vec{x} = B\vec{v}$, thus iff $\vec{x} \in \ker((M, \vec{b})) = \Lambda$. 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 $NV S_{\vec{b}}$ is equal to 0 modulo \vec{b}_S , *i.e.*, each column of $V S_{\vec{b}}$ belongs to Λ . Hence, there exists $R \in \mathbb{Z}^{n \times n}$ such that $V S_{\vec{b}} = (U^{-1} S) R$, *i.e.*, $S_{\vec{b}} = V^{-1} U^{-1} S R$. Using $S_{\vec{b}} = U_1 B U_2$, we get that $B = PSQ$ for P and Q in $\mathbb{Z}^{n \times n}$. From [14, Lemma 1] we conclude that \vec{b} must have more unit entries than \vec{s} . Therefore, (U_p, \vec{s}_p) requires the fewest array dimensions. ■

Recall the equivalent modular mappings $i \bmod 2$ (which is the one built for the kernel $2\mathbb{Z}$) and $2i \bmod 4$ and note that the latter's storage usage is a multiple of the former's.

B. Strictly admissible integer lattices

Let K be an arbitrary set in \mathbb{R}^n . A lattice is **strictly admissible** for K if it does not contain a point other than $\vec{0}$ in K . The quantity $\inf\{d(\Lambda) \mid \Lambda \text{ strictly admissible for } K\}$ is the **critical determinant** $\Delta(K)$ of K . We also define the quantity we are interested in: $\Delta_{\mathbb{Z}}(K) = \inf\{d(\Lambda) \mid \Lambda \subseteq \mathbb{Z}^n \text{ strictly admissible for } K\}$. While $\Delta(K)$ may not be attained, $\Delta_{\mathbb{Z}}(K)$ is an integer and, thus, there always exists a strictly admissible integer lattice Λ such that $d(\Lambda) = \Delta_{\mathbb{Z}}(K)$. Thanks to Propositions 1 and 2, we have:

Proposition 3: 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 an array of $d(\Lambda)$ memory locations.

Proposition 3 affords us dual views of the allocation problem. In Section IV-A, we shall build a strictly admissible integer lattice for \mathcal{D} and deduce a valid mapping. In Section IV-B, we build the modular mapping directly, working with the matrix M and the vector \vec{b} . In the remainder of the paper, we focus on the crux of the problem, the study of $\Delta_{\mathbb{Z}}(K)$ for a 0-symmetric convex body K with positive volume, *i.e.*, when the set of conflicting differences DS is represented or approximated by a set $\mathcal{D} = \overset{\circ}{K}$, the integer

points in K . We seek mechanisms that are robust enough to handle well even extreme cases, for example obtained with “skewed” schedules as in [17]. For computational reasons, we assume that K is a rational *polytope*, *i.e.*, defined by integer linear inequalities. Also, to simplify the presentation of the heuristics and their performance, we assume that K contains n linearly independent integer points. This allows us to talk about the volume $\text{Vol}(K)$ 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 if maybe positive, could be arbitrarily small. In the general case, we can make a preliminary change of basis (at least conceptually) and work in the smallest vector subspace that contains all integer points in K .

C. Lower bounds on $\Delta_{\mathbb{Z}}(K)$ and Minkowski’s first theorem

Many theoretical results exist for $\Delta(K)$ (see [18] for an extensive study), but it is an open problem to be able to compute $\Delta(K)$ for most bodies K . We can wonder if the combinatorial nature of $\Delta_{\mathbb{Z}}(K)$ (integer lattices instead of general lattices in \mathbb{R}^n) can make the problem easier. But, as Proposition 4 shows (for a proof see [14, Proposition 5]), for large bodies, computing $\Delta(K)$ and $\Delta_{\mathbb{Z}}(K)$ are equivalent problems.

Proposition 4: For a bounded star body ² K , $\lim_{\lambda \rightarrow \infty} \frac{\Delta_{\mathbb{Z}}(\lambda K)}{\Delta(\lambda K)} = 1$.

Our goal is then to adapt to $\Delta_{\mathbb{Z}}(K)$ well-known lower and upper bounds for $\Delta(K)$. Recall the simple lower bound in Equ. (1), Section II-C.2. In terms of strictly admissible lattices, this means that for any set S such that $S - S \subseteq K$, $\Delta_{\mathbb{Z}}(K) \geq \text{Card}(S)$. We may argue thus for 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 $\overset{\circ}{K} - \overset{\circ}{K} \subseteq K - K = 2K$. Thus, $d(2\Lambda) \geq \text{Card}(\overset{\circ}{K})$ and $d(\Lambda) \geq \text{Card}(\overset{\circ}{K})/2^n$. It follows that $\Delta_{\mathbb{Z}}(K) \geq \text{Card}(\overset{\circ}{K})/2^n$.

We can get similar inequalities using volumes, thanks to Minkowski’s first theorem [19], which states that if Λ is a lattice of \mathbb{R}^n and K a 0-symmetric bounded convex body with $\text{Vol}(K) > 2^n d(\Lambda)$, K contains a nonzero lattice point of Λ . Thus, if Λ is strictly admissible for K , $d(\Lambda) \geq \text{Vol}(K)/2^n$. This shows that $\Delta_{\mathbb{Z}}(K) \geq \text{Vol}(K)/2^n$. This may be applied to our original problem, *i.e.*, in terms of the set of conflicting indices. Let $S = \overset{\circ}{K}_S$ for a polytope K_S such that $K_S - K_S \subseteq K$ where $\mathcal{D} = \overset{\circ}{K}$. Then $\Delta_{\mathbb{Z}}(K) \geq \text{Vol}(K_S - K_S)/2^n$ and, because of Brunn-Minkowski theorem (see [18, p. 12]), we get $\Delta_{\mathbb{Z}}(K) \geq \text{Vol}(K_S)$.

²A star body is a closed set K such that λx is in the interior of K , for all $x \in K$ and $0 \leq \lambda < 1$. A full-dimensional convex region is a star body, so Proposition 4 applies to K such that $\mathcal{D} = \overset{\circ}{K}$ with our assumptions.

For an upper bound, we could hope to adapt the Minkowski-Hlawka theorem [20], which states that $\Delta(K) \leq \text{Vol}(K)$. But the construction relies on rational lattices and it is not clear whether it can be generalized to $\Delta_{\mathbb{Z}}(K)$. However, in Section IV we build strictly admissible lattices Λ with $d(\Lambda)/\text{Vol}(K)$ upper bounded by a function of n only.

D. Optimal construction

Here we use an oracle that determines the strict admissibility for K of a candidate lattice Λ . When K is a polytope this may be done with integer linear programming; alternatively, if all elements of $\overset{\circ}{K}$ can be enumerated, we can solve integer linear systems to check whether Λ intersects K only in $\vec{0}$. Given such an oracle, $\Delta_{\mathbb{Z}}(K)$ can be found by exhaustive search of candidate lattices. Indeed, thanks to Hermite form (see [15]), we can generate all integer lattices with given determinant d by generating all nonnegative upper triangular matrices, with determinant d , and such that $h_{i,j} < h_{i,i}$ for $j > i$. The number $H_n(d)$ of lattices that need to be generated is equal to $\prod_{j=1}^{n-1} (\pi^{k+j} - 1) / (\pi^j - 1)$ when $d = \pi^k$ for a prime π , and if $d = pq$, p and q relatively prime, then $H_n(d) = H_n(p)H_n(q)$ (see [15, Thm. II.4]). If we find no strictly admissible lattice with determinant d , we continue the search with the value $d+1$. The procedure will stop with an optimal solution having determinant less than $\text{Vol}(K)$ multiplied by a function of n .

Though not practical for bodies with large volumes and high dimensions, this search can be implemented for small sets \mathcal{D} and it will give a \mathcal{C} -valid linear allocation that requires the least possible storage, $\Delta_{\mathbb{Z}}(\mathcal{D})$. This is how we find the optimal modular allocation in the case study introduced in Section I. With the original 4D problem formulation, the number of lattices from $d = 1$ to the optimal $d = 112$ is already large, equal to 86416644, and checking them all with integer linear programming (ILP) to find an optimal solution takes roughly 2 days. With the 3D representation however (see details in Section VII), we generate the 941901 lattices and checked them with ILP in roughly 30 minutes. We did not make any effort yet to try to speed up this exhaustive search.

IV. MEMORY ALLOCATION HEURISTICS

We propose heuristics that build strictly admissible integer lattices Λ for a given polytope K . We arrive at lattices such that $d(\Lambda)/\text{Vol}(K)$ is upper bounded by a function of n only, *i.e.*, at heuristics that are *optimal up to a multiplicative factor* (see Section III-C). These heuristics are all based on a scaling scheme. We start with a set of n linearly

independent integer vectors, and the corresponding lattice (or its dual set) is then scaled to a strictly admissible lattice. As we are going to see, several techniques may be used for determining valid scaling factors. Using the correspondence of Proposition 3, the heuristics are given either in terms of strictly admissible lattice or modular mapping constructions. Their qualities are essentially established by considering sufficiently small scaling factors. Indeed these factors determine the determinant of the lattice, hence the size of the allocation (Proposition 3).

For basic notions and results on convex bodies, dual sets, and successive minima, please refer to [18]. Recall that $K \subset \mathbb{R}^n$ is a 0-symmetric closed bounded convex body, which contains n linearly independent integer vectors. In the following, we identify points and vectors, and for a set A of vectors we denote by $\text{Vect}(A)$ the corresponding vector space.

A. Using the successive minima

Appropriate vectors and scaling factors may be constructed using the central notion of successive minima of K with respect to \mathbb{Z}^n (we adapt to $\Delta_{\mathbb{Z}}(K)$ a technique that Rogers developed for $\Delta(K)$, see [18, Thm. 18.1]). The function $F(\vec{x}) = \inf\{\lambda > 0 \mid \vec{x} \in \lambda K\}$ defines a norm such that $F(\alpha\vec{x}) = |\alpha|F(\vec{x})$, called the **gauge function** of K . The set K is the set of points whose norm is less than or equal to one.

For $1 \leq i \leq n$, the **i -th minimum** $\lambda_i(K)$ of K with respect to \mathbb{Z}^n is the smallest value λ such that at least i linearly independent integer points satisfy $F(x) \leq \lambda$, *i.e.*, $\lambda_i(K) = \inf\{\lambda > 0 \mid \dim(\text{Vect}(\lambda K \cap \mathbb{Z}^n)) \geq i\}$. Since K contains n independent integer points we have $\lambda_i(K) \leq 1$, for $1 \leq i \leq n$. Note that for a point $\vec{x} \neq \vec{0}$ in K and $\mu > 1/F(\vec{x})$, the point $\mu\vec{x}$ is outside K . The principle of the heuristic below is to use scaling factors greater than the $1/\lambda_i(K)$ in obtaining a strictly admissible lattice.

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 $i < n$, and $\dim(\mathcal{L}_i) \leq i - 1$, where $\mathcal{L}_i = \text{Vect}(K/\rho_i \cap \mathbb{Z}^n)$.
- Choose a basis $(\vec{a}_1, \dots, \vec{a}_n)$ of \mathbb{Z}^n such that $\mathcal{L}_i \subseteq \text{Vect}(\vec{a}_1, \dots, \vec{a}_{i-1})$.
- Define Λ as the lattice generated by the vectors $(\rho_i \vec{a}_i)_{1 \leq i \leq n}$.

Theorem 2: The lattice Λ computed by Heuristic 1 is strictly admissible for K . If each ρ_i is the smallest possible power of 2, then $d(\Lambda) \prod_{i=1}^n \lambda_i(K) \leq 2^n$, and $d(\Lambda) \leq n! \text{Vol}(K)$.

The proof of Theorem 2 for upper bounding the product of the ρ_i 's and the size of the allocation relies on an upper bound for $\prod_{i=1}^n 1/\lambda_i(K)$ that involves $\text{Vol}(K)$ (see [14, Thm. 3]), which shows how the geometry of K is taken into account in our process.

Example 1 (Cont'd): The 2 successive minima of K (see Fig. 4) are $\lambda_1(K) = \lambda_2(K) = 1$. With $\rho_1 = \rho_2 = 2$ in 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 when $\vec{b} = (2, 2)$. Actually, this was obvious since nonzero vectors in DS are not multiple of 2, while $U\vec{x} \bmod \vec{b} = 0$ iff $\vec{x} = 2U^{-1}\vec{y}$ for some integer vector \vec{y} , which is $\ker(\sigma) = 2\mathbb{Z}^2$. \square

B. Heuristics based on gauge functions

The previous construction scales a basis whose axes depend on the successive minima. The next construction may be preferred for computational aspects, it is based on “successive widths” of K hence does not require the computation of the minima.

Successive widths of K are defined through $K^* = \{\vec{y} \in \mathbb{R}^n \mid \vec{y} \cdot \vec{x} \leq 1 \text{ for all } \vec{x} \in K\}$, the **dual body**, where $\vec{y} \cdot \vec{x}$ is the inner product of \vec{y} and \vec{x} . For example, the unit ball scaled to have radius r has, as its dual, the unit ball scaled to radius $1/r$; the dual of the “unit-rectangle” $\{z \mid |z_i| \leq 1\}$ is the generalized diamond shape (an octahedron in 3D) $\{z \mid \sum |z_i| \leq 1\}$. With our assumptions on K , we have $(K^*)^* = K$.

Using duality (see [21] and its annotated version [22]), the gauge function of K^* is $F^*(\vec{y}) = \inf\{\lambda > 0 \mid \vec{y} \in \lambda K^*\} = \sup\{\vec{y} \cdot \vec{x} \mid \vec{x} \in K\}$. Furthermore, for n linearly independent integer vectors $(\vec{c}_1, \dots, \vec{c}_n)$, one can define K_i^* the projection of K^* along $\vec{c}_1, \dots, \vec{c}_{i-1}$ into $\text{Vect}(\vec{c}_i, \dots, \vec{c}_n)$. In other words, $\vec{y} = \alpha_i \vec{c}_i + \dots + \alpha_n \vec{c}_n \in K_i^*$ iff there are $\alpha_1, \dots, \alpha_{i-1}$ such that $\alpha_1 \vec{c}_1 + \dots + \alpha_{i-1} \vec{c}_{i-1} + \vec{y} \in K^*$. The gauge function associated to K_i^* is defined by $F_i^*(\vec{y}) = \sup\{\vec{y} \cdot \vec{x} \mid \vec{x} \in K, \vec{c}_1 \cdot \vec{x} = \dots = \vec{c}_{i-1} \cdot \vec{x} = 0\}$, $1 \leq i \leq n$.

For a geometrical intuition of Heuristic 2 below, note that $\sup\{\vec{y} \cdot \vec{x} \mid \vec{x} \in K\} = F_1^*(\vec{y}) = F^*(\vec{y})$ can be interpreted as half of the width of K in the direction \vec{y} . We use scaling factors $\rho_i > F_i^*(\vec{c}_i)$ for constructing lattice points outside K .

Heuristic 2:

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

This heuristic generalizes the successive modulo principle of Lefebvre and Feautrier [3] (see Section V-B) to any set of n linearly independent integer vectors.

Theorem 3: The kernel Λ of the modular mapping (M, \vec{b}) built by Heuristic 2 is strictly admissible 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 $F_i^*(\vec{c}_i) \geq 1$ for all $1 \leq i \leq n$, *i.e.*, the successive widths of K in the directions \vec{c}_i are more than 1, then $d(\Lambda) \leq (n!)^2 \text{Vol}(K)$.

Here the quality of the heuristic is established from the facts that $F_i^*(\vec{c}_i) \geq 1$, for all i , and that $\prod_{i=1}^n F_i^*(\vec{c}_i)$ is upper bounded (see the proof in [14]). These conditions indicates how the choice of $(\vec{c}_1, \dots, \vec{c}_n)$ must depend on the geometry of K^* and K . Vectors such that the successive widths are “not too small” and “not too big” should be preferred.

This may be relaxed by pre-computing an appropriate basis from any given basis $(\vec{c}_1, \dots, \vec{c}_n)$ of \mathbb{Z}^n . We may use the **generalized basis reduction** of [21]. Given a basis $(\vec{c}_i)_{1 \leq i \leq n}$ for \mathbb{Z}^n , the reduction algorithm of [21] outputs a reduced basis $(\vec{r}_i)_{1 \leq i \leq n}$ for K^* , which is then used in Heuristic 2. According to [21, Thm. 3], a reduced basis satisfies:

$$\lambda_i(K^*)(1/2 - \epsilon)^{i-1} \leq F_i^*(\vec{r}_i) \leq \lambda_i(K^*)(1/2 - \epsilon)^{i-n} \text{ for } 0 < \epsilon < 1/2. \quad (2)$$

With $\epsilon = 1/4$, this leads to a mapping of size $d(\Lambda) \leq (n+1)2^{n(n-2)}(n!)^2 \text{Vol}(K)$ [14].

Furthermore, if similarly to Heuristic 1 the successive minima of K^* are available, we may build an *ad hoc* basis $(\vec{c}_1, \dots, \vec{c}_n)$ of \mathbb{Z}^n leading to a guaranteed performance $d(\Lambda) \leq (n!)^2 \text{Vol}(K)$. We may also demonstrate a *guaranteed bounding box mechanism* (as defined in Section V-A, Page 21) for K on particular basis [14].

Example 1 (Cont'd): Working in K^* amounts to computing the successive widths of the set K depicted in Fig. 4. 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, $(i \bmod 2, j \bmod 2)$. 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 2 has no guaranteed performance for this particular basis. But we see from Equ. (2) 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 $Ni + j \bmod 2$. \square

From a practical point of view, we point out that Heuristic 2 can be slightly modified to define a valid 1D mapping [14]. There is no obvious such mechanism for Heuristic 1.

Corollary 1: If (M, \vec{b}) is a valid modular mapping built by Heuristic 2, the 1D modular mapping defined by $(\vec{c}_1 + b_1\vec{c}_2 + b_1b_2\vec{c}_3 + \dots + b_1 \cdots b_{n-1}\vec{c}_n) \cdot \vec{i} \bmod \prod_{i=1}^n b_i$ is also valid.

In [14], we investigate several other aspects of the heuristic design. We study in particular a heuristic dual to Heuristic 2 (*i.e.*, working directly in K), and we propose a polynomial-time heuristic based on the LLL lattice basis reduction [23].

V. A SURVEY OF THE LITERATURE

In this section, we describe in more detail the papers mentioned in Section I to highlight their underlying concepts and illustrate their limitations. The goal of this paper is indeed to be able to a) understand why/when these approaches work/fail, b) possibly derive better solutions and/or handle more general cases.

A. De Greef, Catthoor, and De Man: memory reduction

In [1], De Greef, Catthoor, and De Man, working in the Atomium project [7], identified the need for memory reduction techniques for embedded multimedia applications. Their method is based on an analysis of the *array addresses* used in the program to be optimized. 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 idea of the intra-array storage optimization, since this is the technique we want to analyze.

First, the schedule of the program is *linearized* in some way (note that this is not always possible), assigning to each operation u a virtual integer time $\theta(u)$. Then, for each array accessed in the program, a **canonical linearization** is chosen. For a n -dimensional array, they consider $2^n n!$ canonical linearizations, $n!$ choices for the order in which dimensions are considered, 2^n choices depending whether each dimension is traversed backwards or forwards. So, for a 2D array A of size (M, N) , there are 8 canonical linearizations: $A(i, j)$ can be mapped in memory to a base address (that we ignore hereafter) plus $\pm Ni \pm j$ or $\pm i \pm Mj$. Then, they compute, for each time t , the maximal address difference $|a_1 - a_2|$, where a_1 and a_2 are two memory addresses that contain a live value at time t , and they define b to be the maximal difference for all possible t . Finally, they wrap the array linearization modulo b , which is indeed correct since two values simultaneously live at a

given time t can never be mapped to the same location. This corresponds to a *window* of b successive addresses that are never shared by values simultaneously live.

Example 1 (Cont'd): To make the discussion shorter, we consider only 2 (among the 8) linearizations of A , the column-major order and the row-major order, and we assume that A is a square array of size N . For the column-major order, $A(i, j)$ is mapped to $i + Nj$ and the maximal address difference is for $t = Ni$ and equal to $N(N - 1) - 1$ (see Fig. 3 and details in [14]). This leads to the mapping $A(i, j) \mapsto (i + Nj) \bmod N(N - 1)$, with only a slight memory reduction compared to the original array. For the row-major order, where $A(i, j)$ is mapped to $Ni + j$, things are much better since mapping and time are “aligned”. At any time t , the addresses t and $t - 1$ contain a live value. We get $b = 2$ and the mapping $A(i, j) \mapsto Ni + j \bmod 2$, which requires only 2 memory cells. \square

In the previous example, the method of De Greef *et al.* finds an optimal solution (with memory size 2). However, this is just because the array addresses are traversed by the schedule exactly as one of the canonical mappings, both in terms of directions and size. For example, if the array A was an array of size $M > N$ instead of N (the loop bounds), the maximal address difference for the row-major order would be $M - N + 1$ (for the two live values $A(i, 0)$ and $A(i - 1, N - 1)$) for a memory size $M - N + 2$ instead of 2. Thus, considering the linearizations of the original array with respect to its full size, even if only part of the array is computed, is a limitation. Also, even if all values of the array are computed, there is no reason (except a practical consideration) for the schedule to be aligned with a canonical linearization of the array, especially after compiler-generated loop transformations or data layout optimizations.

Later, aware of some of these limitations, Tronçon, Bruynooghe, Janssens, and Catthoor (also from the Atomium project) proposed in [8] to compute in the original n -dimensional space of array indices, a n -dimensional *bounding box* (*i.e.*, n moduli computed separately as the maximal index address difference in each dimension, defining a n -dimensional rectangular window), instead of a 1D window (*i.e.*, the modulo b) in the linearized space of addresses. We will not discuss further this bounding box mechanism since it is subsumed by Lefebvre and Feautrier *successive modulo* approach, explained in the next section.

B. Lefebvre and Feautrier: storage management

Lefebvre and Feautrier in [3], concerned with automatic parallelization of static control programs, developed a technique of partial data expansion, which (even if not the original

goal) can also be used for memory reduction. Like De Greef, *et al.*, they use inter-array optimization (based on graph coloring). We focus here on the heart of their approach, their intra-array storage optimization. They completely ignore the array to which a statement writes in the original program. They instead first rewrite the program so that each statement S , surrounded by n loops, now writes to a n -dimensional dedicated array A_S . More precisely, each operation $u = (S, \vec{i})$ writes to array element $A_S(\vec{i} \bmod \vec{b})$, where \vec{b} is a positive integer vector of dimension n (thus a particular modular mapping (M, \vec{b}) with M the identity matrix). The corresponding memory size is $\prod_i b_i$.

Given a statement S , the components of \vec{b} are computed as follows. Lefebvre and Feautrier first compute the (lexicographically) maximal time “delay” $D(S)$ between the write at operation (S, \vec{i}) and its last read. While they do not formulate it exactly this way, they consider that \vec{i} and \vec{j} are conflicting when what they call the **utility spans** $[\theta(S, \vec{i}), \theta(S, \vec{i}) + D(S)]$ and $[\theta(S, \vec{j}), \theta(S, \vec{j}) + D(S)]$ intersect. This, in general, is a super-approximation of the relation \bowtie (and the sets CS and DS) we introduced in Section II-B. When these time utility spans do not intersect, (S, \vec{i}) and (S, \vec{j}) can indeed store their results in the same memory location since one of these results is dead before the other one is computed. Then, each b_i is computed successively: b_1 is set to 1 plus the maximal difference $j_1 - i_1$ for all \vec{i}, \vec{j} such that $\vec{i} \bowtie \vec{j}$. With this choice, all operations (S, \vec{i}) and (S, \vec{j}) such that $\vec{i} \bowtie \vec{j}$ will write in different locations since they do not even write in the same “row” (*i.e.*, first dimension) of the array A_S , except possibly when $j_1 = i_1$. These remaining conflicting iterations are handled with the other dimensions: b_2 is set to 1 plus the maximal difference $j_2 - i_2$ for all \vec{i}, \vec{j} such that $\vec{i} \bowtie \vec{j}$ and $j_1 = i_1$. The process goes on until all \vec{i}, \vec{j} such that $\vec{i} \bowtie \vec{j}$ are 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 the De Greef *et al.* approach, dimensions are not considered in their $n!$ possible orders: the successive moduli are always computed from the outermost to the innermost loop of the original program.

This approach subsumes the bounding box approach proposed in [8], which defines b_k as the maximal difference $j_k - i_k$ among all \vec{i}, \vec{j} such that $\vec{i} \bowtie \vec{j}$, but without the constraints $j_1 = i_1, \dots, j_{k-1} = i_{k-1}$. Note that this **successive modulo** mechanism is exactly our Heuristic 2 in Section IV-B, but in the particular case where the basis $(\vec{c}_1, \dots, \vec{c}_n)$ is given by the indices of the original program. Let us come back now to our running example.

Example 1 (Cont'd): Note that with the technique of Lefebvre and Feautrier, the fact that, in the original program, the array is accessed as $A(i, j)$ is irrelevant. If A was defined as a 1D 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 since only the writing and reading iterations are of importance.

With Fig. 3, we see that the successive modulo approach defines $b_1 = 2$, *i.e.*, 1 plus the maximal difference for the first dimension between two conflicting iterations. Then, it remains to consider all conflicting iterations with same loop counter i and the maximal difference in terms of the loop counter j : we get $b_2 = 2$, for the mapping $(S, \vec{i}) \mapsto A(i \bmod 2, j \bmod 2)$ with size 4, a bit worse, but of same order as what we found before. Note that our Corollary 1 shows that the mapping $(S, \vec{i}) \mapsto A(i + 2j \bmod 4)$ is also valid.

We could do the same study with the equivalent 2D schedule given Page 10. But now if, for this schedule, we define the *utility spans* with the quantity $D(S)$ explained previously, this over-constrains the problem and leads to 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 Fig. 3 on the right): b_1 is still equal to 2, but now any two indices whose difference is less than $(1, 1 - N)$ are considered to be conflicting, 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!

Now, suppose that the code of Fig. 2 is scheduled, not as written, but with the schedule $i + Nj$ for the first loop and $i + Nj + 1$ for the second (*i.e.*, apply a loop permutation $(i, j) \mapsto (j, i)$ on each loop, and initiate the second loop 1 clock cycle after the first one). The conflicting iterations are the same as those depicted on Fig. 3, except that the 2 axes should be permuted. Now we get $b_1 = N$, then $b_2 = 2$ for a memory size $b_1 b_2 = 2N$. Here the successive modulo mechanism is applied with the axes in the wrong order! \square

One could argue that, here, the two problems we revealed come maybe from the fact that the schedule θ , either one-dimensional with a *parameter* N , or only *piecewise* affine multidimensional, is not really considered in Lefebvre and Feautrier model. This is true, but, still, this reveals two aspects: for the first problem, it means that for such “pipelined” codes, there is, even before a mapping problem, a problem to model the notion of “time” since the approximation with $D(S)$ has not the same effect whether we use a 1D schedule or a 2D schedule. For the second problem, one could propose to try all $n!$ permutations of the axes as De Greef *et al.* do. This is indeed very likely to work in practice. As we have seen in Section IV, Lefebvre and Feautrier successive modulo mechanism is very robust,

but for extreme cases (e.g., see Example 2 in Section VI), to apply it, our contribution reveals that it is important to choose both the basis and the order in which dimensions are considered. One should not always rely, as they do, on one particular basis and one particular order of dimensions, those given by the original program.

C. Quilleré and Rajopadhye: memory usage optimization

In [2], Quilleré and Rajopadhye studied the problem of memory reuse for systems of recurrence equations, a computation model used to represent simple algorithms to be compiled into circuits. They propose to use an approach similar to Lefebvre and Feautrier approach, *i.e.*, placing the result of an equation S for vector \vec{i} (a concept similar to an operation (S, \vec{i}) for a program) in a dedicated array A_S , but in array element $A_S(f(\vec{i}))$, where f is a linear access function. Their goal is to find an affine function f into a linear space of *smallest* dimension (what they call a “projection”), independent on the parameters such as N in the previous examples. The number of dimensions they obtain is linked to the *depth* $d \geq 1$ of $D(S)$, *i.e.*, one plus the number of leading zeros in $D(S)$. Compared to the two previous approaches, it is a bit more complicated to explain it in a short paragraph without losing accuracy (see their original paper [2], or [14] for more details). We can just say that, basically, their approach exploits the fact that conflicting indices $\vec{i} \bowtie \vec{j}$ lie in a subspace of dimension $n - d + 1$ (and, in general, “thin” in one dimension) to define a $(n - d + 1)$ -dimensional mapping as a projection onto this subspace (plus a modulo 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.

They also mention the use of moduli in all dimensions, but only with a brief analysis and more with a bounding box mechanism in mind as in [8] than with a successive modulo mechanism as in [3]. So, in our opinion, their main contribution was rather to show the interest to work, not necessarily in the basis of the original loop indices (as Lefebvre and Feautrier do), but in a different basis, and in particular in a basis built from the schedule. We point out however that to apply Quilleré and Rajopadhye 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 2 is “hidden”, because this is the linearized version of the schedule $\theta(i, j) = (i, j)$. This is typically the case for pipelined codes scheduled with very skewed linearized schedules as

those developed in [17]. Also, Quilleré and Rajopadhye 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, 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. This has been our direction of work in Section IV.

D. Early work on parallel memories and templates

Linear allocation is not new: it was introduced in 1971 by Budnik and Kuck [24] who were concerned with memory **skewing schemes** for defining data layout allowing parallel accesses to memory by codes that make repeated accesses to a pattern of nearby data elements. Budnik and Kuck called these **templates**; they are also known as stencils. This concept was studied in more detail by Shapiro [25], then Wijshoff and van Leeuwen (see in particular [26]), and others. The validity constraints for this purpose are different than for memory reuse. This perhaps explains why this work on linear allocation has been forgotten in the previously discussed papers (including ours [5]). We think it important, however, to briefly present the similarities and differences between the two contexts.

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. A template T is a finite subset $T = \{\vec{0}, \vec{t}_1, \dots, \vec{t}_l\}$ of \mathbb{Z}^n and an instance $T(\vec{x})$ of T , $\vec{x} \in \mathbb{Z}^n$, is $T(\vec{x}) = \{\vec{x}, \vec{x} + \vec{t}_1, \dots, \vec{x} + \vec{t}_l\}$. We 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 when for any $\vec{x} \in \Lambda$, the restriction of s to $T(\vec{x})$ is an injection. For a *linear* skewing scheme, one can also define a set of conflicting differences $DT = \{\vec{i} - \vec{j} \mid \vec{i}, \vec{j} \in T\}$. Then, a skewing scheme s is (T, \mathbb{Z}^n) -valid iff $\ker(s) \cap DT = \{\vec{0}\}$ as in Proposition 1, which shows a correspondence between linear allocations and linear skewing schemes. Our optimization techniques can be used to derive linear skewing schemes valid for templates expressed as or approximated by polytopes. Conversely, we may hope to reuse results of the theory of linear skewing schemes when \mathcal{D} is equal to the difference set of a template T , *i.e.*, $\mathcal{D} = T - T$. Unfortunately, the main results of this theory are for templates of *small* sizes, and the theory focuses on the mathematical properties of such allocations (properties that have inspired us in Section III) but not

on the *algorithmic* aspects of how to build them. Indeed, since a template is small and described by extension, it was considered to be easy to find the optimal allocation (see [27] and Section III hereafter). This is not satisfying for us when the size of \mathcal{D} can be large, when it is specified by some representation whose size does not depend on the number of points it contains (e.g., polytope), and when it is parameterized. However, one spectacular result developed at this time [28] 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 [29], lower and upper bounds on the optimal allocation are studied, for several complex template shapes, but only in dimension 1, while we mainly work in higher dimensions.

We may also point out that the theory of skewing schemes considers nonlinear allocations (e.g., *diamond schemes* [30] and *multiperiodic schemes* [31]), and a more general notion of validity based on template collections, but still, no practical investigations that we could apply to nonlinear allocations. In conclusion, beyond the connection between memory allocations and skewing schemes, the validity constraints, the size and representation of index sets, and complexity issues are quite different.

VI. GENERAL DISCUSSION

Our allocation constructions rely both on the choice (or computation) of a working basis, and on scaling. Our main contribution is to show that the resulting allocation size can be guaranteed for particular bases. Conversely, a bad choice of basis may lead to an allocation size arbitrarily large with respect to the volume of K . For instance on the following variant of Example 1, we ensure a constant allocation size, while previous heuristics may lead to sizes of order more than N .

Example 2: We keep the code of Ex. 1 but with the iteration domain of Fig. 6 and traversed sequentially with successive diagonals, *i.e.*, with the multi-dimensional schedule $(i - j, i)$, the second loop still initiated one clock cycle later. We get the set DS in Fig. 7, with vertices $(1, 1)$, $(N - 1, N)$, and their opposites. The Lefebvre-Feautrier heuristic would use the basis $\{(1, 0), (0, 1)\}$ and choose $\rho_1 = N + 1$, $\rho_2 = 1$, with a memory size of order N . The technique of De Greef *et al.* loses 2 orders of magnitude (size of order N^2). But the mapping $(i \bmod 2, j \bmod 2)$ is still valid (found by Heuristic 1 for example). \square

In addition to revealing that the choice of the basis is a key point for performance, our

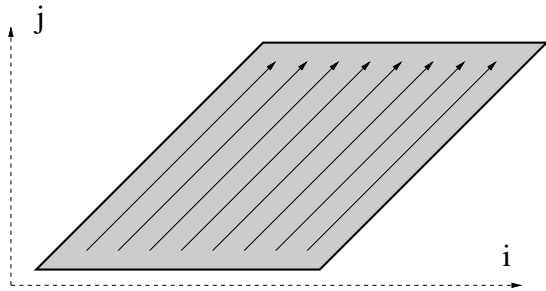
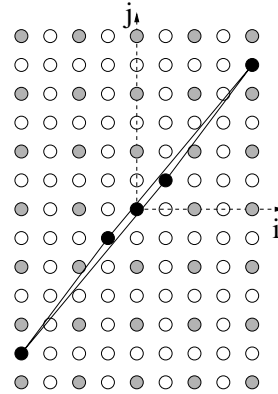


Fig. 6. Iteration domain for Example 2.

Fig. 7. Corresponding set DS .

study helps understanding properties of a “good” working basis. At the cost of knowing the successive minima, or a reduced basis, we are always able to compute an adequate choice. Applying this strategy to the body of Example 2 overcomes the difficulties that lead other heuristics to sizes in N or N^2 , and allows us to find an allocation of size $O(1)$.

However, it may be prohibitive to rely on successive minima or generalized basis reduction. Therefore, it is essential to also point out that previously known heuristics may lead to satisfying allocations, in practice. In Section V, we have seen that these heuristics choose bases (or linearizations) related either to the array indices (De Greef *et al.*), to the loop indices of the initial program (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 . In practice, for hand-written programs, access functions to arrays are simple, scheduling functions (*i.e.*, loop transformations) are not too complicated, and the (possibly sub-)domain where iterations write in memory is not too skewed, 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, and heuristics can work.

Furthermore, if the loops are scheduled with a multi-dimensional *sequential* schedule $\theta(\vec{i}) = (\vec{c}_1 \cdot \vec{i}, \dots, \vec{c}_n \cdot \vec{i})$, one can consider the vectors \vec{c}_i to be a good basis for a heuristic in K^* . It is indeed 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 (see in [2] the fact that DS is flat for the dimensions given by the depth, and in general contains some nonzero integer vectors for all remaining dimensions). In that case, our study shows that mixing the techniques

of Quilleré and Rajopadhye [2] – for choosing the adequate basis – and of Lefebvre and Feautrier [3] – for choosing the modulo vector guarantees the performance. For more complex cases our approach would first identify the subspace in which DS lies, then apply Heuristic 2 in a well-chosen basis.

Example 2 (Cont'd): For the set DS in Fig. 7, if we follow the schedule, *i.e.*, consider the basis $\{\vec{c}_1, \vec{c}_2\} = \{(1, -1), (1, 0)\}$, we retrieve the set DS of Fig. 4 and get the mapping $(i - j \bmod 2, i \bmod 2)$. Actually, we even saw (Heuristic 1) that $\rho_1 = \rho_2 = 2$ is valid whatever the basis, thus the lattice depicted as grey points in Fig. 7 is also valid. \square

Concerning the way moduli are computed, Heuristic 1 first computes the moduli (the ρ_i 's), then a suitable basis for these moduli. Heuristic 2 follows the successive modulo technique of Lefebvre and Feautrier. The technique of De Greef *et al.* is to work with a particular linearization and compute a unique modulo. When this linearization is good, performance is better. Hence an alternative solution could be to first try to reduce the dimension of the problem, then to compute the moduli (see Section VII).

Actually, we do obtain linearizations with Heuristic 2. Indeed, we derive in Corollary 1 a 1D modular mapping, *i.e.*, a linear access function with a single modulo, with guaranteed performance. This is important in practice for limiting the cost of the access function. If we do not use the derived mapping but linearize blindly the multi-dimensional array of size $\prod_i b_i$, then, for example in 2D, from a mapping $(i, j) \mapsto (i \bmod b_1, j \bmod b_2)$ obtained by a successive modulo approach, we could 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 simplest form $(i, j) \mapsto \text{base_address} + (i + b_1j) \bmod b_1b_2$, which is also valid (though not equivalent) and saves a modulo. Note also that, in all heuristics we developed, we can choose moduli that are powers of 2, which in practice may also be important.

We conclude this general discussion by treating an example for a general set K . The following 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. However, 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 and apply Heuristic 2. For this example, Heuristic 1 always gives a good mapping, while Heuristic 2 gives a good mapping 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.

Example 3: Consider the code of Ex. 1, but traversed with the schedule $\theta(i, j) =$

$(i + j, j)$ for the first loop, the second loop starting 1 clock cycle later again. Fig. 8

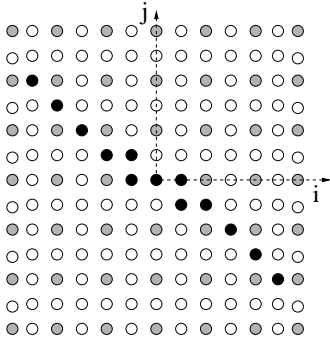


Fig. 8. The set DS in the original basis.

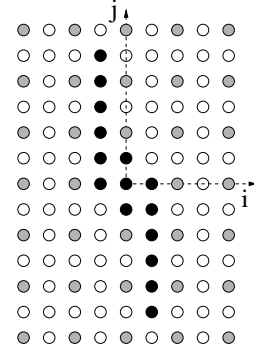


Fig. 9. The set DS in the basis given by the schedule.

gives DS in the original basis: it contains $(1, -1)$, all vectors $(p, p - 1)$ for $0 \leq p < N$, and their opposites. Heuristic 1 shows that, in any basis, the modulo vector $(2, 2)$ is valid since $\dim(DS) = 2$, but $\dim(\lambda DS) = 0$ for $\lambda < 1$ (*i.e.*, $\lambda_1(DS) = \lambda_2(DS) = 1$). Thus the mapping $(i \bmod 2, j \bmod 2)$ is valid (its kernel is in grey). The successive modulo approach (Heuristic 2) 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 (see Fig. 9), we get $b_1 = b_2 = 2$, for the mapping $(i + j \bmod 2, j \bmod 2)$, which is equivalent to $(i \bmod 2, j \bmod 2)$. \square

VII. 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 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 the code given in the introduction, Fig. 1). Here, to make things simpler, we assume that each operation of S writes 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 [4], taking into account the available resources, in particular load-store units, possibly leading to different ρ 's for each micro-statement.)

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 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 $(\delta_{br}, \delta_{bc}, \delta_r, \delta_c)$ satisfying the following inequalities (with $\rho = 8$):

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

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 $8 \times (64 \times b_r + b_c)$ and 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 $8 \times (64 \times b_r + b_c) + 7$, 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_r 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 in dimension 4 but, for clarity, it is better represented as the union of five 2-dimensional parts. The central part corresponds to the particular values $\delta_{br} = \delta_{bc} = 0$ and is depicted in Fig. 10. 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 Fig. 11 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 5 pieces form the whole set K . To get yet a better view of the set K , consider again the set depicted in Fig. 4. 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.

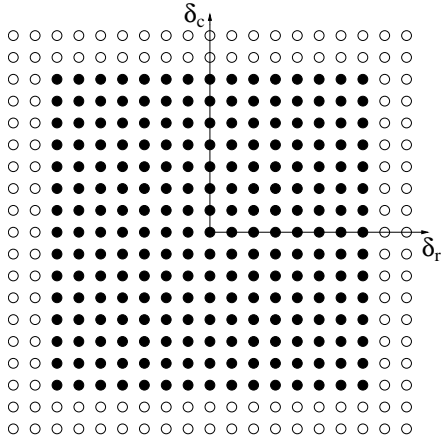


Fig. 10. K part for $(\delta_{br}, \delta_{bc}) = (0, 0)$.

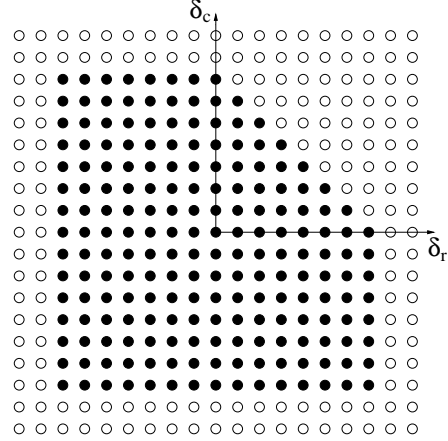


Fig. 11. K part for $(\delta_{br}, \delta_{bc}) = (0, 1)$ and $(\delta_{br}, \delta_{bc}) = (1, -63)$.

We can now derive modular allocations. If we apply Heuristic 2 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 Fig. 10 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 would find since, here, we picked the basis given by the schedule, which is also the basis of the original code. However, if the original code was 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 would consider the index b_c first and find successively $\rho_{bc} = 64$, $\rho_{br} = 1$, $\rho_r = 8$, and $\rho_c = 8$, for a memory size equal to 4096. Again, the choice of basis, and the order in which we evaluate the F_i^* 's for Heuristic 2 is important.

Following our linearization mechanism for Heuristic 2 (Corollary 1), 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 could 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 Fig. 10 for $t = 0$, the part of Fig. 11 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 could 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 *et al.* would have found 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 *et al.* would even 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 III-D, 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 (see the number of lattices given in Section III-D). We find that the smallest possible determinant is 112 and that there are two equivalence classes (*i.e.*, two lattices) that achieve this: in 4D, these are the lattice generated by the vectors $(1, 0, 0, 12)$, $(0, 1, 0, 12)$, $(0, 0, 4, 20)$, $(0, 0, 0, 28)$ and, due to the symmetry in c and r , the lattice generated by the vectors $(1, 0, 0, 20)$, $(0, 1, 0, 20)$, $(0, 0, 4, 12)$, $(0, 0, 0, 28)$. We finally get a corresponding mapping with minimal dimension thanks to Theorem 1. For example, for the first lattice, we can write (Smith form):

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -12 & -12 & -5 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 12 & 12 & 20 & 28 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 28 \end{pmatrix}$$

and, from the left matrix, we get the allocation $(r \bmod 4, -12(b_r + b_c) - 5r + c \bmod 28)$, with many other equivalent mappings such as $(r \bmod 4, 16(b_r + b_c) + 2r + c \bmod 28)$. This is a 2D allocation with no equivalent 1D version. The best 1D 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 $64t + 8r + 3c \bmod 113$.

VIII. CONCLUSION

Several important questions remain, both from theoretical and practical viewpoints.

With our assumptions, restricting to 1D mappings (which are important in practice) is still 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). But how much exactly do we lose with 1D mappings as opposed to multi-dimensional mappings? For the example in Section VII, we only lose 1 cell.

Can we better exploit the fact that the set of conflicting differences \mathcal{D} may not be any 0-symmetric convex body K , but *comes from a scheduled program*? We already mentioned that an “often-quite-good” strategy in the case of a sequential multi-dimensional affine schedule and when the mapping is expressed in terms of loop indices is to mix the two techniques of [3] and [2], *i.e.*, build the moduli as Lefebvre and Feautrier do, but in the basis of the schedule as done by Quilleré and Rajopadhye. Can we identify classes of programs, *i.e.*, give reasonable assumptions, where 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 $n! \text{Vol}(K)$), and in practice? In particular, maybe there are better heuristics based on the construction of a strictly admissible (possibly *rational*) lattice that could 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 be the optimal modular allocations 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?

A complete study remains to be done to better understand the impact of the different approximations used to build \mathcal{D} and the link with dependences and schedules. In particular, it is also important to be able to handle not only polytopes, but a union of polytopes. This paper is just the first step to address all these questions: we only focused on the mathematical framework and its general properties. Future work needs to address more specific theoretical questions as well as strategies for practical implementations.

DEDICATION

This problem arose from our work on the HP Labs PICO Project, conceived and led by Bob Rau. We consider ourselves very lucky to have known Bob as a coworker and a friend. His qualities of curiosity, humor, amiability, and tirelessness were extraordinary, and made it a great pleasure to work beside him. We fondly dedicate this work to his memory.

ACKNOWLEDGMENT

The numerical results for the case study of Section VII were generated automatically by the tool Cl@k (Critical Lattice Kernel) developed by Fabrice Baray, who implemented the algorithms and heuristics presented in this paper. We thank him greatly for his help.

REFERENCES

- [1] E. De Greef, F. Catthoor, and H. De Man, "Memory size reduction through storage order optimization for embedded parallel multimedia applications," *Parallel Computing*, vol. 23, pp. 1811–1837, 1997.
- [2] F. Quilleré and S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 5, pp. 773–815, 2000.
- [3] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, pp. 649–671, 1998.
- [4] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman, "PICO: Automatically designing custom computers," *IEEE Computer*, vol. 35, no. 9, pp. 39–47, Sept. 2002.
- [5] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," in *6th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'03)*, San Jose, USA, Oct. 2003, pp. 298–308.
- [6] Synfora, <http://www.synfora.com>.
- [7] F. Catthoor et al., "Atomium: A toolbox for optimising memory I/O using geometrical models," <http://www.imec.be/design/atomium/>.
- [8] R. Tronçon, M. Bruynooghe, G. Janssens, and F. Catthoor, "Storage size reduction by in-place mapping of arrays," in *Verification, Model Checking and Abstract Interpretation, Third International Workshop, VMCAI 2002*, ser. LNCS, A. Cortesi, Ed., vol. 2294. Springer Verlag, 2002, pp. 167–181.
- [9] B. Kienhuis, E. Rijpkema, and E. 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, USA, May 2000.
- [10] A. Turjan and B. Kienhuis, "Storage management in process networks using the lexicographically maximal preimage," in *14th International Conference on Application-specific Systems, Architectures and Processors (ASAP'03)*. The Hague, The Netherlands: IEEE Computer Society Press, June 2003.
- [11] P. Quinton et al., "Alpha homepage: A language dedicated to the synthesis of regular architectures," <http://www.irisa.fr/cosi/ALPHA>.

- [12] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe, "A unified framework for schedule and storage optimization," in *International Conference on Programming Language Design and Implementation (PLDI'01)*. ACM Press, 2001, pp. 232–242.
- [13] M. M. Strout, L. Carter, J. Ferrante, and B. Simon, "Schedule-independent storage mapping for loops," in *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. San Jose, USA: ACM Press, 1998, pp. 24–33.
- [14] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," LIP, ENS-Lyon, <http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-23.ps.gz>, Tech. Rep. RR2004-23, Apr. 2004.
- [15] M. Newman, *Integral Matrices*. Academic Press, 1972.
- [16] A. Darte, M. Dion, and Y. Robert, "A characterization of one-to-one modular mappings," *Parallel Processing Letters*, vol. 5, no. 1, pp. 145–157, 1996.
- [17] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien, "Constructing and exploiting linear schedules with prescribed parallelism," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 1, pp. 159–172, 2002.
- [18] P. M. Gruber and C. G. Lekkerkerker, *Geometry of Numbers*, 2nd ed. North Holland, 1987.
- [19] J. C. Lagarias, "Point lattices," in *Handbook of Combinatorics*, R. Graham, M. Grötschel, and L. Lovász, Eds. Elsevier Science Publishers B.V., 1995, vol. I, ch. 19, pp. 919–966.
- [20] P. M. Gruber, "Geometry of numbers," in *Handbook of Convex Geometry*, P. Gruber and J. Wills, Eds. Elsevier Science Publishers B.V., 1993, vol. B, ch. 3.1, pp. 739–763.
- [21] L. Lovász and H. E. Scarf, "The generalized basis reduction algorithm," *Mathematics of Operations Research*, vol. 17, no. 3, pp. 751–764, 1992.
- [22] L. Hafer, "The generalized basis reduction algorithm (annotated)," June 2000, <http://www.cs.sfu.ca/~lou/MITACS/grb.pdf>.
- [23] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, pp. 515–534, 1982.
- [24] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Transactions on Computers*, vol. C-20, pp. 1566–1569, Dec. 1971.
- [25] H. D. Shapiro, "Theoretical limitations on the efficient use of parallel memories," *IEEE Transactions on Computers*, vol. C-27, no. 5, pp. 421–428, May 1978.
- [26] H. A. G. Wijshoff and J. Van Leeuwen, "The structure of periodic storage schemes for parallel memories," *IEEE Transactions on Computers*, vol. C-34, no. 6, pp. 501–505, June 1985.
- [27] —, "Periodic storage schemes with a minimum number of memory banks," Rijksuniversiteit Utrecht, the Netherlands, Tech. Rep. RUU-CS-83-4, Feb. 1983.
- [28] —, "Periodic versus arbitrary tessellations of the plane using polyominoes of a single type," Rijksuniversiteit Utrecht, the Netherlands, Tech. Rep. RUU-CS-82-11, July 1982.
- [29] G. Tel, J. Van Leeuwen, and H. A. G. Wijshoff, "The one dimensional skewing problem," Rijksuniversiteit Utrecht, the Netherlands, Tech. Rep. RUU-CS-89-23, Oct. 1989.
- [30] W. Jalby, J.-M. Frailong, and J. Lenfant, "Diamond schemes: An organization of parallel memories for efficient array processing," INRIA, Centre de Rocquencourt, Tech. Rep. 342, 1984.
- [31] G. Tel and H. A. G. Wijshoff, "Hierarchical parallel memory-systems and multi-periodic skewing schemes," Rijksuniversiteit Utrecht, the Netherlands, Tech. Rep. RUU-CS-85-24, Aug. 1985.