Monoparametric Tiling of Polyhedral Programs

Guillaume Iooss, Christophe Alias, Sanjay Rajopadhye

RESEARCH REPORT
N° 9233
December 2018
Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9233--FR+ENG

Monoparametric Tiling of Polyhedral Programs

Guillaume Iooss[*], Christophe Alias[†], Sanjay Rajopadhye[‡]

Project-Team Cash

Research Report n° 9233 — December 2018 — 28 pages

Abstract: Tiling is a crucial program transformation, adjusting the ops-to-bytes balance of codes to improve locality. Like parallelism, it can be applied at multiple levels. Allowing tile sizes to be symbolic parameters at compile time has many benefits, including efficient autotuning, and run-time adaptability to system variations. For polyhedral programs, parametric tiling in its full generality is known to be non-linear, breaking the mathematical closure properties of the polyhedral model. Most compilation tools therefore either perform fixed size tiling, or apply parametric tiling in only the final, code generation step. We introduce monoparametric tiling, a restricted parametric tiling transformation. We show that, despite being parametric, it retains the closure properties of the polyhedral model. We first prove that applying monoparametric partitioning (i) to a polyhedron yields a union of polyhedra, and (ii) to an affine function produces a piecewise-affine function. We then use these properties to show how to tile an entire polyhedral program. Our monoparametric tiling is general enough to handle tiles with arbitrary tile shapes that can tesselate the iteration space (e.g., hexagonal, trapezoidal, etc). This enables a wide range of polyhedral analyses and transformations to be applied.

Key-words: Tiling, Program Transformation, Polyhedral Model

[*] CNRS, ENS de Paris, Inria
[†] Univ Lyon, Inria, CNRS, ENS de Lyon, Université Claude Bernard Lyon 1, LIP UMR 5668, F-69007 LYON, France
[‡] Colorado State University

Tuilage monoparamétré de programmes polyédriques

Résumé : Le tuilage est une transformation de programmes très utilisée en parallélisation automatique. Le tuilage permet notamment de distribuer un calcul sur des unités parallèles tout en réglant le ratio calculs/communications. Tout comme le parallélisme, le tuilage peut être appliqué sur plusieurs niveaux. En gardant des tuiles de taille paramétrée, on permet à chaîne de compilation polyédrique de raisonner analytiquement sur la taille des tuiles. Lorsque ces paramètres survivent à la compilation, on permet l'optimisation du programme à l'installation ou à l'exécution. Pour des programmes polyédriques, le tuilage paramétré est connu pour être non-linéaire dans le cas général, ce qui en fait une transformation non-polyédrique, et rend impossible la composition avec d'autres transformations polyédriques. Nous montrons que lorsque le tuilage ne dépend que d'un paramètre, il redevient polyédrique et possède ainsi les bonnes propriétés de clotûre pour être intégré à une chaine de compilation polyédrique. Notre tuilage monoparamétré est assez général pour traiter des tuiles définies par n'importe quel polyèdre convexe (hexagone, trapèze, etc), ce qui permet une grande variété de transformations polyédriques.

Mots-clés : Tuilage de boucles, transformation de programmes, modèle polyédrique

1 Introduction

In the exascale era, multicore processors are increasingly complicated. Programming them is a challenge, especially when seeking the best performance, because we need to simultaneously optimize conflicting factors, notably parallelism and data locality, and that too, at multiple levels of the memory/processor hierarchy (e.g., at vector-register, and at core-cache levels). Indeed, the very notion of "performance" may refer to execution time (i.e., speed) or to energy (product of the average power and time), or even the energy-delay product.

To tackle these challenges, a *domain specific* mathematical formalism called the *polyhedral model* [38, 37, 14, 15, 16] has energed over the past few decades. A polyhedral program is one a program gas an "iteration space" that is the union of polyhedra, and where memory accesses are affine functions of the iteration vector. For such programs, the model provides a compact, mathematical representation of both programs and their transformations.

Among these program transformations, iteration space tiling [24, 49, 30] (also called loop blocking [42] or partitioning [11, 12, 44]) is a critical transformation, used for multiple objectives: balancing granularity of communication to computation across nodes in a distributed machine, improving data locality on a single node, controlling locality and parallelism among multiple cores on a node, and, at the finest grain, exploiting vectorization while avoiding register pressure on each core. It is an essential strategy used by compilers and automatic parallelizers, and also directly by programmers. As the name suggests, tiling "blocks" iterations into groups (called *tiles*) which are then executed atomically.

One of the key properties of the tiling transformation concerns the nature of the tile sizes. If they are constant, we have *fixed-size tiling* and the transformed program remains polyhedral, albeit with (typically) double the number of dimensions. This means that we can continue to apply further polyhedral analyses and transformations, but because tile sizes are fixed at compile time, any modification of tile sizes necessitates re-generation and recompilation of the program, which takes time. Pluto [10, 1] is a state-of-the-art polyhedral source-to-source compiler that currently applies up to two levels of fixed-size tiling.

If the tile sizes are symbolic parameters, we have a *parametric tiling*. Because the tile sizes are chosen after compile time, we can perform a tile size exploration (commonly used as part of an auto-tuning step [18, 34, 47]) without having to recompile. However, the program is no longer polyhedral after transformation, thus no subsequent polyhedral analysis or transformation can be applied. Thus, parametric tiling is usually the last transformation applied to a program, and is hard-wired in the code generator. This forces the compilation strategy after tiling to be non-polyhedral, and sacrifices modularity. D-tiler [25, 41] and P-tile [7] are state-of-the-art parametric tiled code generators.

In this paper, we introduce a new kind of tiling, called *monoparametric tiling*. As the name suggests, it uses a single tile size parameter: all tile sizes are multiples of this parameter. In other words, the "aspect-ratio" of a tile is fixed. For example, if we consider a rectangular 2-dimensional tile shape and b a program parameter, $2b \times b$ will be a monoparametric tiling, but $b \times b'$ will not be one.

We will prove the closure properties of monoparametric tiling: a polyhedral program transformed by such a tiling remains polyhedral. This allows us to retain all advantages of fixed-size tiling, while providing partial parametrization of tile sizes. Our contributions are as follows.

- We show that monoparametric partitioning transformation[1] is a polyhedral transformation. In particular, we show that the reindexing of the indices of a polyhedron can be expressed as a finite union of Presburger set.
- Likewise, we show that monoparametrically partitioning the two spaces of an affine function transforms it into a piecewise affine function.

[1]We use the convention that while a *tiling* transformation must preserve legality, a *partitioning* is simply a reindexing of the original program. This allows us to reasoned about partitionings purely mathematically, without extraneous constraints.

- Using these two main results, we show how to apply monoparametric partitioning and tiling transformation to a polyhedral program,[2] and also study its scalability. We support any polyhedral tile shape.
- We implemented the monoparametric partitioning transformation on polyhedra and affine functions as both a standalone tool[3] written in C++, and also in the *AlphaZ* polyhedral compiler [51]. We also integrated the standalone library inside a polyhedral source-to-source `C` compiler[4]. We compare the efficiency of monoparametrically tiled code with the corresponding fixed-size and fully parametric code, all produced by the same polyhedral compiler and with similar transformation parameters.

We start in Section 2 by introducing, the background notions needed in the rest of this paper. In Section 3, we focus on the monoparametric partitioning transformation, and prove that this is a polyhedral transformation by studying its effect on a polyhedron and an affine function. In Section 4, we present some scalability results of the monoparametric partitioning transformation on SARE and compare the monoparametric tiled code with the fixed-size tiled code. We describe the related work in Section 5, before concluding in Section 6.

2 Background: Polyhedral Compilation and Tiling

The *polyhedral model* [38, 37, 14, 15, 16] is an established framework for automatic parallelization and compiler optimization. It is used to quantitatively reason about, and systematically transform a class of data- and compute-intensive programs. It may be viewed as the technology to map (i.e., "compile" in the broadest sense of the word) high level descriptions of domain-specific programs to modern, highly parallel processors and accelerators. It abstracts the iterations of such programs as integral points in polyhedra and allows for sophisticated analyses and transformations: exact array dataflow analysis [14], scheduling [15, 16], memory allocation [13, 35] or code generation [36, 8]. To fit the polyhedral model, a program must satisfy conditions that will be described later.

A *polyhedral compiler* is usually a source-to-source compiler, that transforms a source program to optimize various criteria: data locality [8], parallelism [16], a combination of locality and parallelism [48, 10] or memory footprint [13] to name a few. The input language is usually imperative (C like) [10, 3]. It may also be an equational language [32, 51]. Today, polyhedral optimizations can also be found in the heart of production compilers [33, 45, 20, 4].

A polyhedral compiler follows a standard structure. A *front-end* parses the source program, identifies the regions that are amenable to polyhedral analysis, and builds an intermediate *polyhedral representation* using array dataflow analysis [14]. This representation is typically an iteration-level dependence graph, where a node represents a polyhedral set of iterations (e.g., a statement and its enclosing loops), and edges represent affine relations between source and destination polyhedra (e.g., dependence functions). Then, *polyhedral transformations* are applied on the representation. Because of the *closure properties* of the polyhedral representation [38, 32] the resulting program remains polyhedral, and transformations can be composed arbitrarily. The choice of the specific transformations optimize various cost metrics or objective functions for various target platforms, and are not the concern of this paper (they are orthogonal to our goal). Finally, a *polyhedral back-end* generates the optimized output program from the polyhedral representation.

In the rest of this section, we present the basic elements of the polyhedral model, starting with polyhedra and affine functions, and continuing with the polyhedral program representation. We finish by explaining the tiling transformation in the context of the polyhedral compilation.

[2]Note that the iteration spaces of a polyhedral program are Presburger sets, a strict generalization of polyhedra.
[3]Available at `https://github.com/guillaumeiooss/MPP`
[4]Online demonstrator available at `http://foobar.ens-lyon.fr/mppcodegen/index.php`

2.1 Polyhedra and Affine Functions

An *affine expression* is an expression of the form $\sum_k a_k.i_k + c$ where the a_k and c are scalars and the i_k are (index) variables. $\sum_k a_k.i_k$ is called the linear part of the expression and c is called the constant part of the expression. An *affine constraint* is an equality or inequality that between an affine expression and zero (or equivalently, between two affine expressions).

In this paper, we will focus on two objects: integer polyhedra and affine functions. An integer *polyhedron* is a set of integer points whose coordinates satisfy a set of affine constraints. An *affine function* is a multi-dimensional function whose symbolic value is an affine expression of its inputs. These two objects are the mathematical building blocks of the *polyhedral model*. A polyhedron is used to represent the set of instances of a statement inside a loop nest (its *iteration domain*). Likewise, an affine function can be used to represent the consumer-producer relationship between two statements (*dependence function*). These objects are used to form a compact representation of the polyhedral program. In the next subsection, we will introduce a program representation based on these concepts.

In addition to the standard indices involved in polyhedra and affine functions, we also allow the affine expressions to manipulate *program parameters*, i.e., constants whose values are known only at the start of the execution of the program (typically, the size of an input array). They are simply treated as "additional" indices.

A polyhedron (resp. an affine function) can be completely described by the matrix of its coefficients, as the following definition shows:

Definition 1 (Polyhedron) *A polyhedron is a set of points of the form: $\mathcal{P} = \{\vec{i} \mid Q\vec{i} + R\vec{p} + \vec{q} \geq \vec{0}\}$, where Q and R are integral matrices, $\vec{q}$ is an integral (index) vector and $\vec{p}$ is a vector of the program parameters.*

In most of this paper, we will consider integral polyhedra, i.e., polyhedra consisting of integral points. Otherwise, we will call such polyhedra *rational polyhedra*.

For example, consider a triangle: $\mathcal{P} = \{i, j \mid i \geq 0,\ j \geq 0,\ i + j \leq N\}$, where N is a parameter. Its matricial representation is:

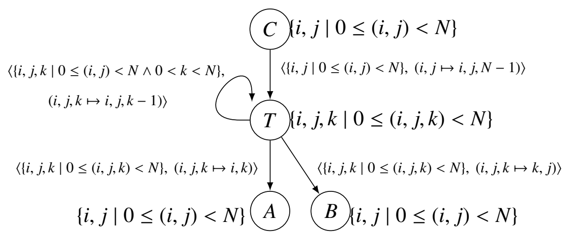
$$\mathcal{P} = \left\{ i, j \;\middle|\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} .(N) + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

Definition 2 (Affine function) *An affine function is a function of the form: $f = (\vec{i} \mapsto A\vec{i} + B\vec{p} + \vec{c})$ where A and B are integral matrices, $\vec{c}$ an integral vector and $\vec{p}$ are the program parameters.*

A *piecewise affine function* is a function defined over a set of disjoint polyhedral domains (called *branches*) and whose definition on each of these branches is (a possibly different) affine function.

Although we earlier stated that the iteration spaces of polyhedral programs are polyhedra, strictly speaking, this is an oversimplification. Actually, these iteration spaces are a generalization of polyhedra called *Presburger sets*, which are defined as sets of the form $\mathcal{P} = \{\vec{i} \mid (\exists \vec{z})\ Q\vec{i} + R\vec{p} + S\vec{z} + \vec{q} \geq \vec{0}\}$. Here, $\vec{z}$ are called *existential variables*. A Presburger set can be viewed as the projection of a polyhedron on the non-existential dimensions. Most modern libraries e.g., the Integer Set Library [46] support Presburger sets and piecewise affine functions.

In Section 3, we present our main results about closure properties for polyhedra and affine functions. Since a Presburger set is simply the image of a polyhedron by a particular form of affine function (specifically, a projection) the extension of the closure properties to Presburger sets follows trivially, as discussed at the end fo that section.


C
$\{i, j \mid 0 \leq (i, j) < N\}$
$\langle\{i, j \mid 0 \leq (i, j) < N\},\ (i, j \mapsto i, j, N-1)\rangle$
$\langle\{i, j, k \mid 0 \leq (i, j) < N \wedge 0 < k < N\},$
$(i, j, k \mapsto i, j, k-1)\rangle$
T
$\{i, j, k \mid 0 \leq (i, j, k) < N\}$
$\langle\{i, j, k \mid 0 \leq (i, j, k) < N\},\ (i, j, k \mapsto i, k)\rangle$
$\langle\{i, j, k \mid 0 \leq (i, j, k) < N\},\ (i, j, k \mapsto k, j)\rangle$
$\{i, j \mid 0 \leq (i, j) < N\}$
A
B
$\{i, j \mid 0 \leq (i, j) < N\}$

$$
\begin{array}{rcl}
C[i, j] & = & 0 \leq (i, j) < N : T[i, j, N-1]; \\
T[i, j, k] & = & \left\{ \begin{array}{l} 0 \leq (i, j) < N, k > 0 : T[i, j, k-1] + A[i, k] \times B[k, j] \\ 0 \leq (i, j) < N, k = 0 : A[i, k] \times B[k, j] \end{array} \right.
\end{array}
$$

Figure 1: Polyhedral Reduced Dependence Graph (left), and the SARE (right) of a matrix multiplication program. In the PRDG,nodes are labeled with a domain $\mathcal{D}$; edges are directed from the consumer to producer, and are labelled with a pair $\langle \mathcal{D}, f \rangle$.

2.2 Program Representation

The most common representation of a polyhedral program is in the form of a *Polyhedral Reduced Dependence Graph* (PRDG). It is a graph in which:

- Each node corresponds to multiple instances of a single generic computation (e.g., a statement) and is labeled with its *iteration space* or *domain* $\mathcal{D}$ (which is a polyhedron).
- Each edge corresponds to a data dependence between a producer and a consumer. It is labeled with a pair, $\langle \mathcal{D}, f \rangle$ where $\mathcal{D}$ is the domain (a, subset of the domain of the consumer node) where the dependence holds, and f is an affine function (associating an index of the consumer with the index of the producer). We use the convention that dependence edges are directed from consumer to the producer.

A PRDG represents dependence information of polyhedral program. For a polyhedral program in a conventional, an imperative language, it can be constructed after performing a dependence analysis [14]. For example, the PRDG of a standard (square, $N \times N$) matrix multiplication program is shown in Figure 1.

In the rest of this paper, we will also use an alternative program representation called *System of Affine Recurrence Equations*, which is nearly equivalent to the PRDG notation. A computation is represented by a list of equations of the form:

$$
\mathtt{Var}[\vec{i}] = \left\{ \begin{array}{l} \ldots \\ \vec{i} \in \mathcal{D}_k : \mathtt{Expr}_k \\ \ldots \end{array} \right.
$$

where each row is a “case branch,” and the $\mathcal{D}_k$ are disjoint polyhedral (sub) domains (called restrictions), and where:

- `Var` is a *variable*, defined over a domain $\mathcal{D}$.
- `Expr` is an *expression*, and can be either:
  - A variable $\mathtt{Var}[\mathtt{f}(\tilde{\mathtt{i}})]$ where `f` is an affine function
  - A constant `Const`,
  - An operation $\mathtt{Op}(\ldots \mathtt{Var}_\mathtt{i}[\mathtt{f}_\mathtt{i}[(\tilde{\mathtt{i}})], \ldots)]$ of arity k (i.e., there are k terms of the form $\mathtt{Var}_\mathtt{i}[\mathtt{f}_\mathtt{i}(\tilde{\mathtt{i}})]$)

In a slightly more general form of SARE, we modify the last clause to $\mathtt{Op}(\ldots \mathtt{Expr}_\mathtt{i}[\mathtt{f}_\mathtt{i}[(\tilde{\mathtt{i}})], \ldots)]$ and furthermore, allow the case branches and restrictions to be arbitrarily nested. This is the representation used in the Alpha language [32, 31].

$$\begin{array}{ll}
C[i,j] = (i,j) \in \mathcal{D}_1 : T[f(i,j)] & \hat{C}[\vec{I}] = \vec{I} \in \hat{\mathcal{D}}_1 : \hat{T}[\hat{f}(\vec{I})] \\
T[i,j,k] = (i,j,k) \in \mathcal{D}_2 : T[g_1(i,j,k)] + A[g_2(i,j,k)] \times B[g_3(i,j,k)] & \hat{T}[\vec{I}] = \vec{I} \in \hat{\mathcal{D}}_2 : \hat{T}[\hat{g}_1(\vec{I})] + \hat{A}[\hat{g}_2(\vec{I})] \times \hat{B}[\hat{g}_3(\vec{I})] \\
T[i,j,k] = (i,j,k) \in \mathcal{D}_3 : A[g_2(i,j,k)] \times B[g_3(i,j,k)] & \hat{T}[\vec{I}] = \vec{I} \in \hat{\mathcal{D}}_3 : \hat{A}[\hat{g}_2(\vec{I})] \times \hat{B}[\hat{g}_3(\vec{I})]
\end{array}$$

Figure 2: Normalized SARE for matrix multiplication (left) and its tiled/partitioned version (right). For convenience, we name the domains and the dependence functions in the left part of Figure 2 as follows. $\mathcal{D}_1 = \{i, j \mid 0 \leq (i, j) < N\}$ and $f(i, j) = (i, j, N-1)$, $\mathcal{D}_2 = \{i, j, k \mid 0 \leq (i, j, k) < N \text{ and } k > 0\}$, $\mathcal{D}_3 = \{i, j, k \mid 0 \leq (i, j, k) < N \text{ and } k = 0\}$, and $g_1(i, j, k) = (i, j, k-1)$, $g_2(i, j, k) = (i, k)$, $g_3(i, j, k) = (k, j)$.

In addition to all the information contained in a PRDG, the SARE/Alpha representation also has specific expressions that evaluated. From a mathematical point of view, because it only has the true dependences, and does not contain any information about scheduling and memory allocation.

We also mention that the Alpha representation, while seemingly more general than SAREs, is mathematically equivalent. Indeed, Mauras [32] proposed a *normalization* procedure (implemented as a program transformation in the AlphaZ system [51]) that systematically "flattens out" any Alpha program into the SARE form.

2.3 Tiling: the essential transformation

Tiling [24, 50] is a program transformation which groups the instances of a loop into sets called *tiles*), such that each tile is atomic. Because of this, we cannot have cyclic dependences between tiles, and this is the essential condition for legality of tiling.

One important aspect of tiling is *tile shape*. The most commonly used shape is a hyper-parallelepiped, defined by its boundary hyperplanes [24, 50]. A particular case is *orthogonal tiling* where the shape is hyper-rectangular—the tile boundaries are normal to the canonic axes. Other shapes have been studied, such as trapezoid (with redundant computation [29]), diamond [6] or hexagonal [39, 19]. Some shapes might have non-integral tile origins, such as diamond tiling formed by a non-unimodular set of boundary hyperplanes.

Another important aspect of the tiling transformation is the *tile size*: tiles can either be of constant size (for example, a 16×32 rectangle), or of parametric size (for example, a rectangular tile of size $b_1 \times b_2$). It is well known that if tile sizes are constant, the transformed program remains polyhedral [24], but for parametric sizes, tiling is no longer polyhedral. To see this, consider orthogonal tiling with tiles of size $b_1, \ldots, b_d$, we have to substitute the original indices by a *quadratic* expression of the form $b_k.i_b + i_l$ where b_k is a program parameter. Thus, the resulting domains and functions are no longer linear/affine, and so, parametric tiling does not respect the closure properties of the polyhedral model.

To illustrate the tiling partitioning and transformation, consider the SARE depicted in Figure 1. To simplify the presentation, we name the domains and the dependence functions shown in the left part of Figure 2 as follows. For equation 1, we have $\mathcal{D}_1 = \{i, j \mid 0 \leq (i, j) < N\}$ and $f(i, j) = (i, j, N-1)$. Similarly, for the other two equations, we have $\mathcal{D}_2 = \{i, j, k \mid 0 \leq (i, j, k) < N \text{ and } k > 0\}$, $\mathcal{D}_3 = \{i, j, k \mid 0 \leq (i, j, k) < N \text{ and } k = 0\}$, and $g_1(i, j, k) = (i, j, k-1)$, $g_2(i, j, k) = (i, k)$, $g_3(i, j, k) = (k, j)$. Now, partitioning the SARE means dividing the index domains (here $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$) into *tiles*. For instance, the left part of Figure 3 shows a possible partitioning of $\mathcal{D}_1$ with size $b \times 2b$ where b is a parameter. This partitioning is defined by a mapping $\mathcal{T}_1 : \mathcal{D}_1 \to \hat{\mathcal{D}}_1, (i, j) \mapsto (i_b, i_l, j_b, j_l)$ from the *original index* $(i, j) \in \mathcal{D}_1$ to its *partitioned index* composed of the *tile coordinates* i_b, j_b (b for "block") and the *offset* in the tile i_l, j_l (l for "local"). The tile index should satisfy: $\mathcal{T}_1^{-1}(i_b, i_l, j_b, j_l) = (b.i_b + i_l, b.j_b + j_l)$.

The right part of Figure 2 depicts the final partitioned SARE. The structure is somehow similar the original SARE. The only differences are the partitioned domains $\hat{\mathcal{D}}_k$ for $k = 1 \ldots 3$ accessing the partitioned variables, and the partitioned index functions $\hat{f}$ and $\hat{g}_k$, $k = 1, 3$. Since the index domains of A, B and C have been modified by the partitioned, the original index functions must be transformed accordingly. For instance, the original index function $g(i, j, k) = (i, j, k-1)$ is transformed to the partitioned index function $\hat{g}(i_b, i_l, j_b, j_l, k_b, k_l) = (i_b, i_l, j_b, j_l, k_b, k_l - 1)$ if $k_l > 0$ and $\hat{g}(i_b, i_l, j_b, j_l, k_b, k_l) = (i_b, i_l, j_b, j_l, k_b - 1, b - 1)$ if $k_l = 0$. In general, the partitioned index function is a piecewise affine function. The source and the target domains can be different, with different dimensions and partitionings (e.g., $\hat{f} : \hat{\mathcal{D}}_1 \to \hat{\mathcal{D}}_2$), which makes it particularly challenging to determine them automatically.

After the partitioning transformation is applied, the partitioned SARE can be *tiled* by a polyhedral backend to produce a tiled sequential program, by simply providing the backend with a schedule and a memory allocation function for each array. The derivation of a tiled schedule is completely orthogonal to our paper, and any state of the art scheduler may be used.

For the matrix multiply example, we may specify the schedule $\theta_C(i_b, i_l, j_b, j_l) = (i_b, j_b, i_l, j_l)$ and $\theta_T(i_b, i_l, j_b, j_l, k_b, k_l) = (i_b, j_b, k_b, i_l, j_l, k_l)$. On the operations defining T, this means that the block indices (i_b, j_b, k_b) are the outer dimensions, and the local indices (i_l, j_l, k_l) are the inner loop/schedule dimensions.

3 Monoparametric Partitioning

In this section, we will focus on monoparametric partitioning. We will first formally define it, and then consider its application to the two base mathematical objects of a polyhedral program: polyhedra and affine functions. We will prove the following closure properties: the monoparametric partitioning of a polyhedron gives a union of Presburger sets and correspondingly, monoparametric partitioning of an affine function yields a piecewise affine function. While these properties hold for any polyhedral tile shape, will we present their specialization to rectangular tile shapes. This is the most common shape, and allows us to greatly simplify the expression of the transformed objects. More general tile shapes can also be handled by appropriate preprocessing (e.g., change-of-basis) transformations.

3.1 Monoparametric partitioning transformation

The partitioning transformation is the reindexing component of the tiling transformation, which introduces the new indices used to express the new schedule. Intuitively, it is a non-affine function which goes from a d-dimensional space to a $2d$-dimensional space. In the case of a non-parametric tile size, this transformation is formalized in the following way:

Definition 3 (Fixed-size partitioning) *We are given a non-parametric bounded convex polyhedron $\mathcal{P}$, called* tile shape*, and a non-parametric rational lattice $\mathcal{L}$, called* lattice of tile origins *with basis, L. Moreover, $\mathcal{P}$ and $\mathcal{L}$ satisfy the* tessellation property*, namely that $\mathbb{Q}^n = \biguplus_{\vec{l} \in \mathcal{L}} \{\vec{l} + \vec{z} \mid \vec{z} \in \mathcal{P}\}$. Then, the* fixed-size partitioning *transformation associated with $\mathcal{P}$ and $\mathcal{L}$ is the reindexing transformation defined by the function $\mathcal{T}$ which decomposes any point $\vec{i}$ in the following way:*

$$\mathcal{T}(\vec{i}) = (\vec{i_b}, \vec{i_l}) \Leftrightarrow \vec{i} = L.\vec{i_b} + \vec{i_l} \quad \textit{where } (L.\vec{i_b}) \in \mathcal{L} \textit{ and } \vec{i_l} \in \mathcal{P}$$

The chosen tile shape affects the nature of the lattice of tile origins: if we have rectangular or diamond partitioning with unimodular hyperplanes, then this lattice must be integral. However, if we consider a diamond partitioning using non-unimodular hyperplanes, this lattice will not be integral.

Now, let us extend the above formalization to the monoparametric case. First, we note that a *homothetic scaling* of a rational set, $\mathcal{D}$, by a constant a, is the set denoted by $a \times \mathcal{D}$, and defined by $a \times \mathcal{D} = \{\vec{z} \mid (\vec{z}/a) \in \mathcal{D}\}$. Using this notion, we define a monoparametric partitioning as simply the homothetic scaling of a fixed-size partitioning by a parametric factor:

Definition 4 (Monoparametric partitioning) *Given a fixed-size partitioning (with tile shape $\mathcal{P}$, lattice of tile origins, $\mathcal{L}$, and reindexing function $\mathcal{T}$), and a fresh program parameter b, called the* tile size parameter*, the* monoparametric partitioning *associated with $\mathcal{P}$, $\mathcal{L}$ and b is a reindexing transformation associated to the function $\mathcal{T}_b$, such that:*

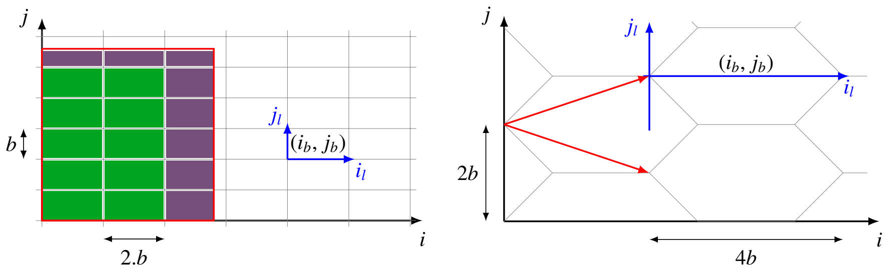


Figure 3: On the left: example of rectangular monoparametric partitioning for the array C of the matrix multiplication. On the right: example of hexagonal monoparametric partitioning for a 2D space. (i_b, j_b) are the block indices, which identify a tile. (i_l, j_l) are the local indices, which identify the position of a point inside a tile. The tile shape is a hexagon with 45° slopes and of size $4b \times 2b$. It can be viewed as the homothetic scaling of a 4×2 hexagon. The red arrows correspond to a basis of the lattice of tile origins.

- *its tile shape is $\mathcal{P}_b = b \times \mathcal{P}$, and*
- *its lattice of tile origins is $\mathcal{L}_b = b \times \mathcal{L}$.*

These two objects form the decomposition function $\mathcal{T}_b$ such that: $\mathcal{T}_b(\vec{i}) = (\vec{i_b}, \vec{i_l}) \Leftrightarrow \vec{i} = b.L.\vec{i_b} + \vec{i_l}$ where $(b.L.\vec{i_b}) \in \mathcal{L}_b$ and $\vec{i_l} \in \mathcal{P}_b$

Exemple. In a two dimensional space, the monoparametric partitioning corresponding to rectangular tiles of sizes $2b \times b$ is defined by the tile shape $\mathcal{P}_b = \{i_l, j_l \mid 0 \leq i_l < 2b \ \wedge \ 0 \leq j_l < b\}$ and the lattice of tile origins $\mathcal{L}_b = b.L.\mathbb{Z}^2$ where $L = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$. The decomposition function is $\mathcal{T}_b(i, j) = (i_b, j_b, i_l, j_l)$ where $i = 2b.i_b + i_l$, $j = b.j_b + j_l$ and $(i_l, j_l) \in \mathcal{P}_b$. Note that i_b and i_l (respectively, j_b and j_l) are the result and modulo associated to the integral division of i by $2b$ (respectively, b). □

Exemple. Figure 3 shows another example of monoparametric partitioning, with hexagonal tiles, defined by the tile shape $\mathcal{P}_b = \{i, j \mid -b < j \leq b \ \wedge \ -2b < i + j \leq 2b \ \wedge \ -2b < j - i \leq 2b\}$ and the lattice of tile origins $\mathcal{L}_b = b.L.\mathbb{Z}^2$ where $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$. The decomposition function $\mathcal{T}_b(i, j) = (i_b, j_b, i_l, j_l)$, where $i = 3b.(i_b + j_b) + i_l$, $j = b.(i_b - j_b) + j_l$ and $(i_l, j_l) \in \mathcal{P}_b$. □

Monoparametric tiling means, applying the monoparametric partitioning as a transformation to a program representation (i.e., an SARE) and this involves three steps: applying it to the domains of the variables, to the dependences, and combining it all. Applying $\mathcal{T}_b$ to a polyhedron $\mathcal{D}$ means computing the image $\mathcal{T}_b(\mathcal{D})$ of $\mathcal{D}$ by $\mathcal{T}_b$. For a dependence function f, there are two spaces to consider: the input and the output spaces. Since each of them may be tiled differently, we have two reindexing functions, one for the input space $\mathcal{T}_b$ and one for the output space $\mathcal{T}'_b$. Thus, applying a monoparametric partitioning transformation to f means computing $\mathcal{T}'_b \circ f \circ \mathcal{T}_b^{-1}$.

We will only consider, without loss of generality, full dimensional partitionings, i.e., where all indices of the considered polyhedron or affine function are replaced by their corresponding tiled and local indices. There is no loss of generality because **partitioning does not imply tiling** but is just the first reindexing step, and the final schedule can be adapted in case some of the reindexed dimensions are not to be tiled. To illustrate this, consider a loop whose indices are (i, j, k), scanning a domain $\mathcal{D}$ in lexicographic order. Consider a rectangular monoparametric tiling transformation $\mathcal{T}_b$. After the partitioning step, we have replaced the original indices by $(i_b, i_l,\ j_b, j_l,\ k_b, k_l)$, belonging to the partitioned iteration domain $\Delta = \mathcal{T}_b(\mathcal{D})$. If we use the lexicographic schedule on these new indices, then the order of the computation will remain unchanged compared to the original loop. We can recover the original indices through the non-linear equality $i = i_b.b + i_l$. If we want to tile the j and k dimensions of this loop, we can permute the loops such that their block indices appear first: $(j_b, k_b,\ i_b, i_l, j_l, k_l)$. We can recover the original indices through the non-linear function $\mathcal{T}_b$. This idea can is also valid for the case of non-rectangular tile shapes.

The goal of this section is to show that monoparametric partitioning is a polyhedral transformation, i.e., that the transformed program, after reindexing, is still a polyhedral program. Because the partitioning only modifies polyhedra and dependence functions of a polyhedral program, it is enough to prove the closure of polyhedron and affine function by this transformation is enough.

3.2 Closure of monoparametric partitioning of polyhedra

We will now show the first of these two results, i.e., that monoparametric partitioning of a polyhedron gives a union of Presburger sets. First, we show this property in the most general framework, which encompass all state-of-the-art tiling transformations, then present the simpler case with rectangular tile shapes and provide some examples.

Theorem 1 (Monoparametric partitioning of a polyhedron) *Given a polyhedron $\mathcal{D} = \{\vec{i} \mid Q.\vec{i} + R.\vec{p} + \vec{q} \geq \vec{0}\}$ where $\vec{p}$ are the program parameters, and a monoparametric partitioning $\mathcal{T}_b$ with tile shape $\mathcal{P}_b$, origin lattice $\mathcal{L}_b$, and size parameter b, then the image: $\mathcal{T}_b(\mathcal{D})$ of $\mathcal{D}$ by $\mathcal{T}_b$ is given by*

$$\mathcal{T}_b(\mathcal{D}) = \bigcap_{0 \leq c < N_{\text{constr}}} \biguplus_{\vec{0} \leq \vec{\alpha} < D.\vec{1}} \left[\biguplus_{k_c^{min} < k_c \leq k_c^{max}} \left\{ \vec{i_b}, \vec{i_l} \mid \begin{array}{c} Q_{c,\bullet}.L.\delta.D^{-1}.\vec{i_b} + \delta.R_{c,\bullet}.\vec{p_b} + (\delta.k_c - Q_{c,\bullet}.L.\delta.D^{-1}.\vec{\alpha}) = 0 \\ \vec{i_b}\ mod\ (D.\vec{1}) = \vec{\alpha} \\ \vec{i_l} \in \mathcal{T}_b \\ b.\delta.k_c \leq \delta.Q_{c,\bullet}.\vec{i_l} + \delta.R_{c,\bullet}.\vec{p_l} + \delta.q_c + b.(Q_{c,\bullet}.L.\delta.D^{-1}.\alpha) \end{array} \right\} \right.$$
$$\left. \uplus \left\{ \vec{i_b}, \vec{i_l} \mid \begin{array}{c} Q_{c,\bullet}.L.\delta.D^{-1}.\vec{i_b} + \delta.R_{c,\bullet}.\vec{p_b} + (\delta.k_c^{min} - Q_{c,\bullet}.L.\delta.D^{-1}.\vec{\alpha}) \geq 0 \\ \vec{i_b}\ mod\ (D.\vec{1}) = \vec{\alpha} \\ \vec{i_l} \in \mathcal{T}_b \end{array} \right\} \right]$$

where:

- N_{constr} *is the number of constraints of* $\mathcal{D}$
- $\vec{p} = \vec{p_b}.b + \vec{p_l}$ *with* $\vec{0} \leq \vec{p_l} < b.\vec{1}$
- $X_{c,\bullet}$ *is the vector corresponding to the c-th row of the matrix X.*
- $\mathcal{L}_b = b.L.D^{-1}.\mathbb{Z}$ *where L is an integral matrix and D is a diagonal integral matrix*
- k_c^{min} *and* k_c^{max} *are constants, depending on the coefficients of the c-th constraint and the tiling chosen.*
- δ *is the smallest common multiplier of the diagonal elements of D.*

Proof.

(Part 1: First decomposition) Let us derive the constraints of $\mathcal{T}_b(\mathcal{D})$ from the constraints of $\mathcal{D}$:

$$Q.\vec{i} + R.\vec{p} + \vec{q} \geq \vec{0} \tag{1}$$

$\mathcal{D}$ is the intersection of N_{constr} half spaces, each one of them defined by a single constraint $Q_{c,\bullet}.\vec{i} + R_{c,\bullet}.\vec{p} + q_c \geq 0$, for $0 \leq c < N_{\text{constr}}$, and we consider each constraint independently. Let us use the definitions of $\vec{i_b}$, $\vec{i_l}$, $\vec{p_b}$ and $\vec{p_l}$ to eliminate $\vec{i}$ and $\vec{p}$.

$$b.Q_{c,\bullet}.L.D^{-1}.\vec{i_b} + Q_{c,\bullet}.\vec{i_l} + b.R_{c,\bullet}.\vec{p_b} + R_{c,\bullet}.\vec{p_l} + q_c \geq 0 \tag{2}$$

Notice that this constraint is no longer linear, because of the $b.\vec{i_b}$ and $b.\vec{p_b}$ terms. To eliminate them, we divide each constraint by $b > 0$ to obtain:

$$Q_{c,\bullet}.L.D^{-1}.\vec{i_b} + R_{c,\bullet}.\vec{p_b} + \frac{Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c}{b} \geq 0$$

In general, this constraint involves rational terms. Thus, in order to obtain integral terms, we take the floor of each constraint (which is valid because $a \geq 0 \Leftrightarrow \lfloor a \rfloor \geq 0$):

$$\left\lfloor Q_{c,\bullet}.L.D^{-1}.\vec{i_b} + R_{c,\bullet}.\vec{p_b} + \frac{Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c}{b} \right\rfloor \geq 0 \tag{3}$$

We consider the modulo of $\vec{i_b}$ in relation to D by considering their integral division: $\vec{i_b} = D.\vec{\lambda} + \vec{\alpha}$ where $\vec{0} \leq \vec{\alpha} < D.\vec{1}$, $\vec{\alpha}$ and $\vec{\lambda}$ being integral vectors. By introducing these quantities into the previous equation, we obtain:

$$Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + \left\lfloor Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + \frac{Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c}{b} \right\rfloor \geq 0 \tag{4}$$

We define $f_c(\vec{\alpha}, \vec{i_l}, \vec{p_l}) = \left\lfloor Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + \frac{Q_{c,\bullet}.\vec{i_l}+R_{c,\bullet}.\vec{p_l}+q_c}{b} \right\rfloor$. Let us show that this quantity can only take a *constant, non parametric* number of values. First, $\vec{\alpha}$ is bounded by constants thus does not cause any issue. We have $\vec{0} \leq \vec{p_l} < b.\vec{1}$. Because the later only appears in the fraction over b and $\vec{0} \leq \frac{\vec{p_l}}{b} < \vec{1}$, $\vec{p_l}$ does not cause any issue.

Finally, we have $\vec{i_l} \in \mathcal{P}_b$. Because $\mathcal{P}_b$ is the homothetic scaling of a parameterless polyhedron $\mathcal{P}$, then $\frac{\vec{i_l}}{b} \in \mathcal{P}$. Thus, we can bound the maximal and minimal values of k_c by constants. Therefore, k_c can only take a constant non parametric number of values, and we have the possibility to create a union of polyhedra, each one of them corresponding to one different value of $f_c(\vec{\alpha}, \vec{i_l}, \vec{p_l})$.

Let us consider an arbitrary value of f_c: $k_c \in [|k_c^{\min}; k_c^{\max}|]$ [5]. Eqn (4) becomes:

$$Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c \geq 0 \tag{5}$$

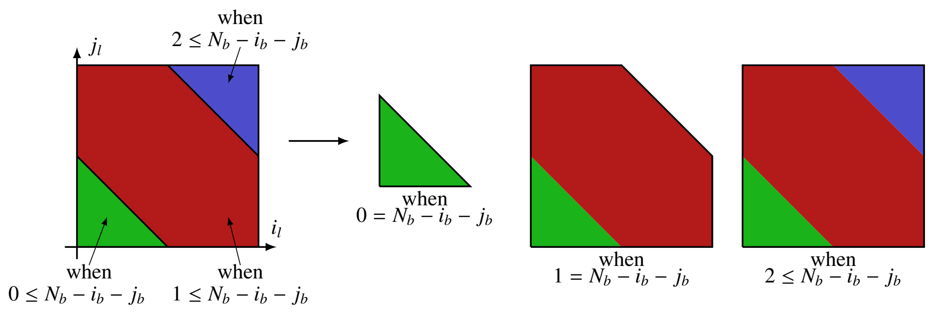
In addition to this constraint, we have the constraints on i_l, p_l and α. Moreover, because we imposed an integer value of f_c, Equation $f_c(\vec{\alpha}, \vec{i_l}, \vec{p_l}) = k_c$ translates to the following affine constraint:

$$b.k_c \leq b.Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c < b.(k_c + 1) \tag{6}$$

To summarize, the c^{th} constraint of (1) corresponds to the union of Presburger sets obtained for each value of k_c:

$$\biguplus_{k_c} \left\{ \vec{i_b}, \vec{i_l} \;\middle|\; \begin{array}{c} Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c \geq 0 \\ b.k_c \leq b.Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c < b.(k_c + 1) \\ \vec{i_l} \in \mathcal{T}_b \\ (\exists \vec{\alpha}, \vec{\lambda})\ \vec{i_b} = D.\vec{\lambda} + \vec{\alpha} \quad \vec{0} \leq \vec{\alpha} < D.\vec{1} \end{array} \right\}$$

[5] $x \in [|a; b|]$ meaning $x \in [a; b]$ and x is an integer



j_l
when $2 \leq N_b - i_b - j_b$
i_l
when $0 \leq N_b - i_b - j_b$
when $1 \leq N_b - i_b - j_b$
when $0 = N_b - i_b - j_b$
when $1 = N_b - i_b - j_b$
when $2 \leq N_b - i_b - j_b$

Figure 4: Stripe coverage of a tile. Given a constraint (e.g. $0 \leq N - i - j$), we have obtain a disjoint union of polyhedra, each polyhedra covering a stripe of a given tile (as shown on the left part of the figure). By examining the constraints on the block indices (e.g. $0 \leq N_b - i_b - j_b + k_c$), we deduce that given a tile, if the stripe k_c occurs in this tile, then all the stripes $k'_c > k_c$ also occurs in this tile.

where k_c enumerates all possible values of $\left\lfloor Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + \frac{Q_c.\vec{i_l}+Q_c^{(p)}.\vec{p_l}+q_c}{b} \right\rfloor$ in the interval $[|k_c^{\min}; k_c^{\max}|]$.

All that remains in order to obtain the partitioning is to intersect these unions for each constraint $c \in [|1; N_{\text{constr}}|]$. However, it is possible to improve the result, as described below.

(Part 2: Reordeirng constraints) First, let us study the pattern of the constraints of the polyhedra of the union. Let us call $(Block_{k_c})$ the constraint on the block indices and $(Local_{k_c})$ the constraints on the local indices. We notice some properties among these constraints (Figure 4):

- Each k_c covers a different stripe of a tile (whose equations is given by $(Local_{k_c})$). The union of all these stripes, for $k_c^{\min} \leq k_c \leq k_c^{\max}$ forms a partition of the whole tile (by definition of $k_c^{\min}$ and $k_c^{\max}$).

- If a tile $\vec{i_b}$ satisfies the constraint $(Block_{k_c})$ for a given k_c, then the same tile also satisfies $(Block_{k'_c})$ for every $k'_c > k_c$ (because $a \geq 0 \Rightarrow a + 1 \geq 0$). In other words, if the k_cth stripe in a tile is non-empty, the tile will have all the k'_c stripes, for every $k'_c > k_c$.

Thus, if a block $\vec{i_b}$ satisfies $(Block_{k_c^{\min}})$, then it satisfy all the $(Block_{k_c})$ for $k_c \geq k_c^{\min}$ and the whole rectangular tile is covered by the union $\mathcal{T}_b(\mathcal{D})$.

Also, if a block $\vec{i_b}$ satisfies exactly $(Block_{k_c})$ (i.e., if $Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c = 0$), then it does not satisfy the $(Block_{k'_c})$ for $k'_c < k_c$ and we do not have the stripes below k_c. Therefore, only the local indices $\vec{i_l}$ which satisfy $(b.k_c \leq b.Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c)$ are covered by the union $\mathcal{T}_b(\mathcal{D})$.

Using these observations, we separate the tiles into two categories: those which satisfy $(Block_{k_c^{\min}})$ (corresponding to a full tile), and those which satisfy exactly a $(Block_{k_c})$ where $k_c^{\min} < k_c$ (corresponding to a portion of the tile).

Mathematically, by splitting all of the polyhedra of the union according to the constraints $Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c = 0$, $k_c^{min} < k_c \leq k_c^{max}$, then pasting them together, we obtain the following improved expression:

$$\bigcap_{0\leq c<N_{\text{constr}}} \left[\left\{ \vec{i_b},\vec{i_l} \;\middle|\; \begin{array}{c} Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c^{min} \geq 0 \\ \vec{i_l} \in \mathcal{T}_b \\ \vec{i_b} = D.\vec{\lambda} + \vec{\alpha}, \quad \vec{0} \leq \vec{\alpha} < D.\vec{1} \end{array} \right\} \biguplus \biguplus_{k_c^{\min}<k_c\leq k_c^{\max}} \left\{ \vec{i_b},\vec{i_l} \;\middle|\; \begin{array}{c} Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c = 0 \\ b.k_c \leq b.Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c \\ \vec{i_l} \in \mathcal{T}_b \\ \vec{i_b} = D.\vec{\lambda} + \vec{\alpha}, \quad \vec{0} \leq \vec{\alpha} < D.\vec{1} \end{array} \right\} \right]$$

Thus, by intersecting all of these unions for each constraint, we obtain the expression of $\mathcal{T}_b(\mathcal{D})$. By distributing the intersection of the union of polyhedra, we obtain a union of disjoint polyhedra. After eliminating empty polyhedra, the number of obtained disjoint polyhedra is the number of different tile shapes of the partitioned version of $\mathcal{D}$.

(Part 3: Eliminating $\vec{\lambda}$ with $\vec{\alpha}$) Finally, let us get rid of $\vec{\lambda}$ and $\vec{\alpha}$ in these constraints. We eliminate $\vec{\lambda}$ by substituting $\vec{\lambda}$ by $(D^{-1}).D.\vec{\lambda}$ which is equal to $D^{-1}.(\vec{i_b} - \vec{\alpha})$. To keep integer values, we introduce δ the smallest common multiplier of the diagonal elements of D, such that $\delta.D^{-1}$ is an integral matrix. We get:

$$\begin{aligned}
Q_{c,\bullet}.L.\vec{\lambda} + R_{c,\bullet}.\vec{p_b} + k_c = 0 \quad &\Leftrightarrow \quad \delta.Q_{c,\bullet}.L.\vec{\lambda} + \delta.R_{c,\bullet}.\vec{p_b} + \delta.k_c = 0 \\
&\Leftrightarrow \quad Q_{c,\bullet}.L.(\delta.D^{-1}).D.\vec{\lambda} + \delta.R_{c,\bullet}.\vec{p_b} + \delta.k_c = 0 \\
&\Leftrightarrow \quad Q_{c,\bullet}.L.(\delta.D^{-1}).(\vec{i_b} - \vec{\alpha}) + \delta.R_{c,\bullet}.\vec{p_b} + \delta.k_c = 0
\end{aligned}$$

In order to eliminate $\vec{\alpha}$, we consider independently each value of $\vec{\alpha}$, in order to form a disjoint union of polyhedra. Notice that each value of $\vec{\alpha}$ corresponds to a different kind of tile origins, thus a different kind of tile shape. The final expression is the following:

$$\bigcap_{0 \le c < N_{\text{constr}}} \biguplus_{\vec{0} \le \vec{\alpha} < D.\vec{1}} \Bigg[\left\{ \vec{i_b}, \vec{i_l} \middle| \begin{array}{c} Q_{c,\bullet}.L.(\delta.D^{-1}).\vec{i_b} + \delta.R_{c,\bullet}.\vec{p_b} + (\delta.k_c^{min} - Q_{c,\bullet}.L.(\delta.D^{-1}).\vec{\alpha}) \ge 0 \\ \vec{i_l} \in \mathcal{T}_b, \quad \vec{i_b} \bmod (D.\vec{1}) = \vec{\alpha} \end{array} \right\}$$
$$\biguplus_{k_c^{\min} < k_c \le k_c^{\max}} \left\{ \vec{i_b}, \vec{i_l} \middle| \begin{array}{c} Q_{c,\bullet}.L.(\delta.D^{-1}).\vec{i_b} + \delta.R_{c,\bullet}.\vec{p_b} + (\delta.k_c - Q_{c,\bullet}.L.(\delta.D^{-1}).\vec{\alpha}) = 0 \\ b.k_c \le b.Q_{c,\bullet}.L.D^{-1}.\vec{\alpha} + Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c \\ \vec{i_l} \in \mathcal{T}_b, \quad \vec{i_b} \bmod (D.\vec{1}) = \vec{\alpha} \end{array} \right\} \Bigg]$$

$\square$

The resulting set is an intersection of unions, which will need to be simplified while progressively eliminating empty polyhedra. For a given constraint, there are as many polyhedra in the union as the number of valid $(\vec{\alpha}, \vec{k_c})$, which is $O(d \times m)$ where $d = \prod_i D_{i,i}$ and $m = max(|Q_{\bullet,\bullet}|, |R_{\bullet,\bullet}|, |q_\bullet|)$. After simplification and removal of empty sets, we obtain in the resulting union of Presburger sets exactly one set per tile shape.

The above theorem is much simpler in the rectangular case, as the following corollary shows:

Corollary 1 *Given a polyhedron $\mathcal{D} = \{\vec{i} \mid Q.\vec{i} + R.\vec{p} + \vec{q} \ge \vec{0}\}$ with size parameters $\vec{p}$, and a rectangular monoparametric partitioning $\mathcal{T}_b$ with shape $\mathcal{P}_b$, tile origin lattice $\mathcal{L}_b$ and size b, then:*

$$\mathcal{T}_b(\mathcal{D}) = \bigcap_{0 \le c < N_{\text{constr}}} \Bigg[\biguplus_{k_c^{min} < k_c \le k_c^{max}} \left\{ \vec{i_b}, \vec{i_l} \mid \begin{array}{c} Q_{c,\bullet}.L.\vec{i_b} + \delta.R_{c,\bullet}.\vec{p_b} + \delta.k_c = 0 \\ \vec{i_l} \in \mathcal{T}_b \\ b.k_c \le Q_{c,\bullet}.\vec{i_l} + R_{c,\bullet}.\vec{p_l} + q_c \end{array} \right\} \bigcup \left\{ \vec{i_b}, \vec{i_l} \mid \begin{array}{c} Q_{c,\bullet}.L.\vec{i_b} + R_{c,\bullet}.\vec{p_b} + k_c^{min} \ge 0 \\ \vec{i_l} \in \mathcal{T}_b \end{array} \right\} \Bigg]$$

where $X_{c,\bullet}$ is the vector corresponding to the c-th row of the matrix X and $\mathcal{L}_b = b.L.\mathbb{Z}$ where L is an integral matrix.

Proof. We apply Theorem 1 with rectangular tiles and an integral lattice of tile origin $\mathcal{L}$. Therefore, $D = Id$, $\delta = 1$ and $\vec{\alpha} = \vec{0}$. This greatly simplifies the resulting expression. $\square$

Exemple. Consider the following polyhedron: $\{i, j \mid j - i \le N \wedge i + j \le N \wedge 0 < j\}$ and the following hexagonal partitioning:

- $\mathcal{P}_b = \{i, j \mid -b < j \le b \;\wedge\; -2b < i + j \le 2b \;\wedge\; -2b < j - i \le 2b\}$
- $\mathcal{L}_b = L.b.\mathbb{Z}^2$ where $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

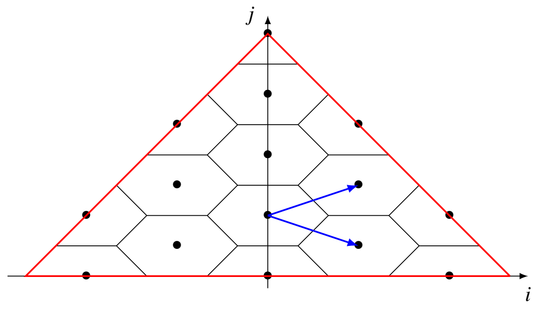


Figure 5: Polyhedron and tiling of Example 3.2. The dots correspond to the tile origins of the tiles contributing to the polyhedron. The blue arrows show the basis of the lattice of tile origins.

For simplicity, we assume that $N = 6.b.N_b + 2b$, where N_b is a positive integer. A graphical representation of the polyhedron and of the tiling is shown in Figure 5.

Let us start with the first constraint of the polyhedron.

$$j-i \leq N \quad \Leftrightarrow \quad 0 \leq 6.b.N_b+2.b+b.(3.i_b+3.j_b)+i_l-b.(i_b-j_b)-j_l \quad \Leftrightarrow \quad 0 \leq 6.N_b+2+2.i_b+4.j_b+\left\lfloor \frac{i_l-j_l}{b} \right\rfloor$$

where $-2b \leq i_l - j_l < 2b$. Therefore, $k_1 = \left\lfloor \frac{i_l-j_l}{b} \right\rfloor \in [|-2,1|]$. For $k_1 = -1$ and 1, the equality constraint $6.N_b + 2.i_b + 4.j_b + 2 + k_1 = 0$ is not satisfied (because of the parity of its terms), thus the corresponding polyhedra are empty.

Let us examine the second constraint of the polyhedron.

$$i+j \leq N \quad \Leftrightarrow \quad 0 \leq 6.b.N_b+2.b-b.(3.i_b+3.j_b)-i_l-L.b.(i_b-j_b)-j_l \quad \Leftrightarrow \quad 0 \leq 6.N_b+2-4.i_b-4.j_b+\left\lfloor \frac{-i_l-j_l}{b} \right\rfloor$$

where $-2b \leq -i_l - j_l < 2b$. Therefore $k_2 = \left\lfloor \frac{-i_l-j_l}{b} \right\rfloor \in [|-2,1|]$. For the same reason as the previous constraint, $k_2 = -1$ and 1 lead to empty polyhedra.

Let us examine the third constraint of the polyhedron.

$$0 \leq j-1 \quad \Leftrightarrow \quad 0 \leq b.(i_b - j_b) + j_l - 1 \quad \Leftrightarrow \quad 0 \leq i_b - j_b + \left\lfloor \frac{j_l-1}{b} \right\rfloor$$

where $-b \leq j_l - 1 < b$. Therefore $k_3 = \left\lfloor \frac{j_l-1}{b} \right\rfloor \in [|-1,0|]$

Therefore, we obtain a union of $2 \times 2 \times 2 = 8$ polyhedra, which are the result of the following intersections:

$$\begin{array}{l}
\left[\begin{array}{l} \{i_b, j_b, i_l, j_l \mid 0 \leq 6.N_b + 2.i_b + 4.j_b \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus\{i_b, j_b, i_l, j_l \mid 0 = 6.N_b + 2.i_b + 4.j_b + 2 \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq i_l - j_l\} \end{array}\right] \\
\cap \left[\begin{array}{l} \{i_b, j_b, i_l, j_l \mid 0 \leq 6.N_b - 4.i_b - 4.j_b \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus\{i_b, j_b, i_l, j_l \mid 0 = 6.N_b - 4.i_b - 4.j_b + 2 \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq -i_l - j_l\} \end{array}\right] \\
\cap \left[\begin{array}{l} \{i_b, j_b, i_l, j_l \mid 0 \leq i_b - j_b - 1 \wedge (i_l, j_l) \in \mathcal{T}_b\} \\ \uplus\{i_b, j_b, i_l, j_l \mid 0 = i_b - j_b \wedge (i_l, j_l) \in \mathcal{T}_b \wedge 0 \leq j_l - 1\} \end{array}\right]
\end{array}$$

□

3.3 Closure of monoparametric partitioning of affine functions

In this subsection, we will prove the second closure property of the monoparametric partitioning transformation, which is on affine function. Again, we will first consider the most general framework before presenting the simpler rectangular case and providing some example.

Theorem 2 (Monoparametric partitioning for affine function) *Given an affine function $f = (\vec{i} \mapsto Q.\vec{i} + R.\vec{p} + \vec{q})$ where $\vec{p}$ are the program parameters, and a monoparametric tiling $\mathcal{T}_b$ (tile shape $\mathcal{P}_b$, tile origin lattice $\mathcal{L}_b$) for the input space and another monoparametric tiling $\mathcal{T}'_b$ (shape $\mathcal{P}'_b$, tile origin lattice $\mathcal{L}'_b$) for the output space and a common tile size parameter b, then, the function compositions $\phi = \mathcal{T}'^{-1}_b \circ f \circ \mathcal{T}_b$ is a piecewise affine function, whose branches have the following form:*

$$\Big(D'.L'^{-1}.Q.L.D^{-1}.\vec{i_b} + D'.L'^{-1}.R.\vec{p_b} + \vec{\alpha'} - D'.L'^{-1}.Q.L.D^{-1}.\vec{\alpha} + D'.L'^{-1}.(\vec{k} - \vec{k'})\ ,$$
$$Q.\vec{i_l} + R.\vec{p_l} + b.(Q.L.D^{-1}.\vec{\alpha} - L'.D'^{-1}.\vec{\alpha'} + \vec{k'} - \vec{k}) + \vec{q}\Big)$$
$$\text{when} \left\{ \begin{array}{c} \vec{i_l} \in \mathcal{P}_b, \quad \vec{i'_l} \in \mathcal{P}'_b \\ \vec{\alpha} = \vec{i_b}\ mod\ \epsilon, \quad \vec{\alpha'} = \vec{i'_b}\ mod\ \epsilon' \\ \vec{k}.b \leq Q.L.D^{-1}.\vec{\alpha}.b + Q.\vec{i_l} + R.\vec{p_l} + \vec{q} < (\vec{k} + \vec{1}).b \end{array} \right.$$

where:

- *$\vec{p} = \vec{p_b}.b + \vec{p_l}$ with $\vec{0} \leq \vec{p_l} < b.\vec{1}$*
- *$\mathcal{L}_b = b.L.D^{-1}.\mathbb{Z}$ where L is an integral matrix and D is a diagonal matrix*
- *$\mathcal{L}'_b = b.L'.D'^{-1}.\mathbb{Z}$ where L' is an integral matrix and D' is a diagonal matrix*
- *ϵ is the smallest integer such that $\epsilon.Q.L.D^{-1}$ is an integral matrix*
- *ϵ' is the smallest integer such that $\epsilon'.L'.D'^{-1}$ is an integral matrix*
- *There is one branch per value of $(\vec{\alpha}, \vec{\alpha'}, \vec{k}, \vec{k'})$, these values being bounded by constants.*

Proof. Let us start from the definition of f: $\vec{i'} = Q.\vec{i} + R.\vec{p} + \vec{q}$. We use the definitions of $\vec{i'_b}$, $\vec{i'_l}$, $\vec{i_b}$, $\vec{i_l}$, $\vec{p_b}$ and $\vec{p_l}$ to eliminate $\vec{i'}$, $\vec{i}$ and $\vec{p}$:

$$L'.D'^{-1}.\vec{i'_b}.b + \vec{i'_l} = Q.(L.D^{-1}.\vec{i_b}.b + \vec{i_l}) + R.(\vec{p_b}.b + \vec{p_l}) + \vec{q} \qquad (7)$$

We would like to consider the modulo of $\vec{i'_b}$ in relation to $D'.L'^{-1}$. However, this last quantity is not integral in general, but a rational matrix. Thus, we introduce ϵ', the smallest common multiple of the denominators of the rational coefficients of $L'.D'^{-1}$, such that $\epsilon'.L'.D'^{-1}$ is integral. We also introduce ϵ, smallest integer such that $\epsilon.Q.L.D^{-1}$ is integral:

$$\left\{ \begin{array}{ll} \vec{i'_b} = \epsilon'.\vec{\lambda'} + \vec{\alpha'}, & \vec{0} \leq \vec{\alpha'} < \epsilon'.\vec{1} \\ \vec{i_b} = \epsilon.\vec{\lambda} + \vec{\alpha}, & \vec{0} \leq \vec{\alpha} < \epsilon.\vec{1} \end{array} \right.$$

By substituting $\vec{i'_b}$ and $\vec{i_b}$ by these equations, we obtain:

$$L'.D'^{-1}.b.(\epsilon'.\vec{\lambda'} + \vec{\alpha'}) + \vec{i'_l} = Q.(L.D^{-1}.b.(\epsilon.\vec{\lambda} + \vec{\alpha}) + \vec{i_l}) + R.(\vec{p_b}.b + \vec{p_l}) + \vec{q}$$
$$\Leftrightarrow b.\epsilon' L'.D'^{-1}.\vec{\lambda'} + b.L'.D'^{-1}.\vec{\alpha'} + \vec{i'_l} = b.\epsilon.Q.L.D^{-1}.\vec{\lambda} + b.Q.L.D^{-1}.\vec{\alpha} + b.R.\vec{p_b} + Q.\vec{i_l} + R.\vec{p_l} + \vec{q}$$

We divide both sides of this last equation by $b > 0$. Then, in order to obtain integral terms, we take the floor of each constraint (which is valid because $[a \geq 0 \Leftrightarrow \lfloor a \rfloor \geq 0]$).

$$\epsilon'.L'.D'^{-1}.\vec{\lambda'} + \left\lfloor L'.D'^{-1}.\vec{\alpha'} + \frac{\vec{i'_l}}{b} \right\rfloor = \epsilon.Q.L.D^{-1}.\vec{\lambda} + R.\vec{p_b} + \left\lfloor Q.L.D^{-1}.\vec{\alpha} + \frac{Q.\vec{i_l}+R.\vec{p_l}+\vec{q}}{b} \right\rfloor$$

We define $\vec{k'}(\vec{\alpha'}, \vec{i'_l}) = \left\lfloor L'.D'^{-1}.\vec{\alpha'} + \frac{\vec{i'_l}}{b} \right\rfloor$ and $\vec{k}(\vec{\alpha}, \vec{i_l}, \vec{p_l}) = \left\lfloor Q.L.D^{-1}.\vec{\alpha} + \frac{Q.\vec{i_l}+R.\vec{p_l}+\vec{q}}{b} \right\rfloor$. Let us show that both quantities can only take a *constant non parametric* number of values. Both $\vec{\alpha'}$ and $\vec{\alpha}$ are bounded by constants. We also have $\vec{0} \leq \vec{p_l} < b.\vec{1}$, thus $\vec{0} \leq \frac{\vec{p_l}}{b} < \vec{1}$. Finally, $\vec{i_l} \in \mathcal{P}_b$ and $\vec{i'_l} \in \mathcal{P}'_b$. Because $\mathcal{P}_b$ and $\mathcal{P}'_b$ are homothetic scaling of parameterless polyhedra $\mathcal{P}$ and $\mathcal{P}'$, then $\frac{\vec{i_l}}{b} \in \mathcal{P}$ and $\frac{\vec{i'_l}}{b} \in \mathcal{P}'$.

Therefore, $\vec{k}$ and $\vec{k'}$ can only take a constant non parametric number of values. Because the value of the resulting piecewise affine function is different for every values of $(\vec{\alpha}, \vec{\alpha'}, \vec{k}, \vec{k'})$, we create one branch for each one of their values. For a specific value of $(\vec{\alpha}, \vec{\alpha'}, \vec{k}, \vec{k'})$, we have:

$$\epsilon'.L'.D'^{-1}.\vec{\lambda'} = \epsilon.Q.L.D^{-1}.\vec{\lambda} + R.\vec{p_b} + \vec{k} - \vec{k'}$$

By substituting this last equality into the equation 7, we obtain the following expression for $\vec{i'_l}$:

$$\vec{i'_l} = (Q.L.D^{-1}.\vec{\alpha} - L'.D'^{-1}.\vec{\alpha'} + \vec{k'} - \vec{k}).b + Q.\vec{i_l} + R.\vec{p_l} + \vec{q}$$

Finally, let us reconstruct $\vec{i_b}$ and $\vec{i'_b}$ while getting rid of $\vec{\lambda}$ and $\vec{\lambda'}$:

$$\begin{array}{rcl}
\vec{i'_b} & = & \epsilon'.\vec{\lambda'} + \vec{\alpha'} \\
& = & D'.L'^{-1}.(\epsilon'.L'.D'^{-1}.\vec{\lambda'}) + \vec{\alpha'} \\
& = & D'.L'^{-1}.(\epsilon.Q.L.D^{-1}.\vec{\lambda} + R.\vec{p_b} + \vec{k} - \vec{k'}) + \vec{\alpha'} \\
& = & D'.L'^{-1}.Q.L.D^{-1}.(\epsilon.\vec{\lambda}) + D'.L'^{-1}.(R.\vec{p_b} + \vec{k} - \vec{k'}) + \vec{\alpha'} \\
& = & D'.L'^{-1}.Q.L.D^{-1}.(\vec{i_b} - \vec{\alpha}) + D'.L'^{-1}.(R.\vec{p_b} + \vec{k} - \vec{k'}) + \vec{\alpha'} \\
& = & D'.L'^{-1}.Q.L.D^{-1}.\vec{i_b} + D'.L'^{-1}.R.\vec{p_b} + \vec{\alpha'} - D'.L'^{-1}.Q.L.D^{-1}.\vec{\alpha} + D'.L'^{-1}.(\vec{k} - \vec{k'})
\end{array}$$

The final expression of one of the branch of the resulting piecewise affine function is the following:

$$(\vec{i'_b}, \vec{i'_l}) = \begin{pmatrix} D'.L'^{-1}.Q.L.D^{-1}.\vec{i_b} + D'.L'^{-1}.R.\vec{p_b} + \vec{\alpha'} - D'.L'^{-1}.Q.L.D^{-1}.\vec{\alpha} + D'.L'^{-1}.(\vec{k} - \vec{k'}) \\ Q.\vec{i_l} + R.\vec{p_l} + b.(Q.L.D^{-1}.\vec{\alpha} - L'.D'^{-1}.\vec{\alpha'} + \vec{k'} - \vec{k}) + \vec{q} \end{pmatrix}$$
$$\text{when} \left\{ \begin{array}{c} \vec{i_l} \in \mathcal{P}_b, \quad \vec{i'_l} \in \mathcal{P}'_b \\ \vec{\alpha} = \vec{i_b} \bmod \epsilon, \quad \vec{\alpha'} = \vec{i'_b} \bmod \epsilon' \\ \vec{k}.b \leq Q.L.D^{-1}.\vec{\alpha}.b + Q.\vec{i_l} + R.\vec{p_l} + \vec{q} < (\vec{k} + \vec{1}).b \\ \vec{k'}.b \leq L'.D'^{-1}.\vec{\alpha'}.b + \vec{i'_l} < (\vec{k'} + \vec{1}).b \end{array} \right.$$

Note that the last constraint is redundant, when substituting $\vec{i'_l}$ by its value, and so, we drop it. □

The resulting piecewise affine function has as many branches as the number of valid $(\vec{\alpha}, \vec{\alpha'}, \vec{k}, \vec{k'})$, which is a $O(d.d'.m)$ where $d = \prod_i D_{i,i}$, $d' = \prod_i D'_{i,i}$ and $m = max(|Q_{\bullet,\bullet}|, |R_{\bullet,\bullet}|, |q_{\bullet}|)$.

In the common case where we use two rectangular partitionings for the input and the output spaces, the expression of the above theorem is greatly simplified:

Corollary 2 *Given an affine function $f = (\vec{i} \mapsto Q.\vec{i} + R.\vec{p} + \vec{q})$, where $\vec{p}$ are the program parameters and given two rectangular monoparametric tiling $\mathcal{T}_b$ (tile shape $\mathcal{P}_b$, lattice of tile origin $\mathcal{L}_b$) for the input space and another monoparametric tiling $\mathcal{T}'_b$ (tile shape $\mathcal{P}'_b$, lattice of tile origin $\mathcal{L}'_b$) for the output space,*

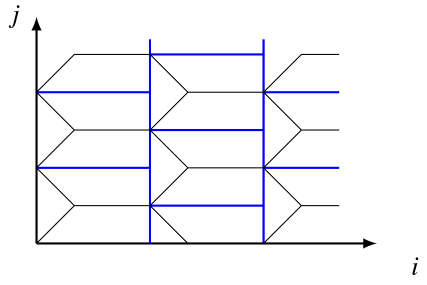


Figure 6: Overlapping of the rectangular (in blue) and the hexagonal tiles in Example 3.3

with a common program parameter b. Then, $\phi = \mathcal{T}_b'^{-1} \circ f \circ \mathcal{T}_b$ is a piecewise affine function, whose value is:

$$\begin{pmatrix} L'^{-1}.Q.L.\vec{i_b} + L'^{-1}.R.\vec{p_b} + L'^{-1}(\vec{k} - \vec{k'}) \\ Q.\vec{i_l} + R.\vec{p_l} + b.(\vec{k'} - \vec{k}) + \vec{q} \end{pmatrix} \text{ when } \begin{cases} \vec{i_l} \in \mathcal{P}_b, \quad \vec{i'_l} \in \mathcal{P}'_b \\ \vec{k}.b \leq Q.\vec{i_l} + R.\vec{p_l} + \vec{q} < (\vec{k} + \vec{1}).b \end{cases}$$

for every value of $(\vec{k}, \vec{k'})$, which are bounded by constants.

Proof. We apply Theorem 2 with rectangular tiles and an integral lattice of tile origin $\mathcal{L}$. Therefore, $D = D' = Id$, $\epsilon = \epsilon' = 1$ and $\vec{\alpha} = \vec{\alpha'} = \vec{0}$. This simplifies greatly the resulting expression. □

Exemple. Consider the identity affine function $(i, j \mapsto i, j)$, and the two following partitioning transformations:

- For the input space, we choose a hexagonal tiling, whose tile shape is $\mathcal{T}_b = \{i, j \mid -b < j \leq b \wedge -2b < i + j \leq 2b \wedge -2b < j - i \leq 2b\}$ and lattice $\mathcal{L}_b = L.b.\mathbb{Z}^2$ where $L = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$
- For the output space, we choose a rectangular tiling, whose tile shape is $\mathcal{T}'_b = \{i, j \mid 0 \leq i < 3b \wedge 0 \leq j < 2b\}$ and lattice $\mathcal{L}'_b = L'.b.\mathbb{Z}^2$ where $L' = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$

An overlapping of these two tilings is shown in Figure 6. The derivation goes as follow:

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \quad \Leftrightarrow \quad L'.b.\begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + \begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = L.b.\begin{bmatrix} i_b \\ j_b \end{bmatrix} + \begin{bmatrix} i_l \\ j_l \end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix} i'_b \\ j'_b \end{bmatrix} + L'^{-1}.\frac{1}{b}.\begin{bmatrix} i'_l \\ j'_l \end{bmatrix} = \begin{bmatrix} i_b \\ j_b \end{bmatrix} + L'^{-1}.\frac{1}{b}.\begin{bmatrix} i_l \\ j_l \end{bmatrix}$$

Because $L'^{-1} = \frac{1}{6}.\begin{bmatrix} 1 & 3 \\ 1 & -3 \end{bmatrix}$, these constraints become:

$$\begin{cases} i'_b + \frac{i'_l + 3.j'_l}{6b} = i_b + \frac{i_l + 3.j_l}{6b} \\ j'_b + \frac{i'_l - 3.j'_l}{6b} = j_b + \frac{i_l - 3.j_l}{6b} \end{cases}$$

After taking the floor of these constraints:

$$\begin{cases} i'_b + \left\lfloor \frac{i'_l + 3.j'_l}{6b} \right\rfloor = i_b + \left\lfloor \frac{i_l + 3.j_l}{6b} \right\rfloor \\ j'_b + \left\lfloor \frac{i'_l - 3.j'_l}{6b} \right\rfloor = j_b + \left\lfloor \frac{i_l - 3.j_l}{6b} \right\rfloor \end{cases}$$

We define $k'_1 = \left\lfloor \frac{i'_l+3.j'_l}{6b} \right\rfloor$, $k_1 = \left\lfloor \frac{i_l+3.j_l}{6b} \right\rfloor$, $k'_2 = \left\lfloor \frac{i'_l-3.j'_l}{6b} \right\rfloor$ and $k_2 = \left\lfloor \frac{i_l-3.j_l}{6b} \right\rfloor$. After analysis of the extremal values of these quantities, we obtain $k_1 \in [|0;1|]$, $k_2 \in [|-1;0|]$, $k'_1 \in [|0;1|]$ and $k'_2 \in [|-1;0|]$.

Therefore, we obtain a piecewise quasi-affine function with 16 branches (one for each value of (k_1, k'_1, k_2, k'_2)). Each branch has the following form:

$$\begin{gathered}\big(i_b + k_1 - k'_1, j_b + k_2 - k'_2,\ \ i_l + 3b(k'_1 + k'_2 - k_1 - k_2), j_l + b(k'_1 + k_2 - k_1 - k'_2)\big) \\ \text{when } 0 \le i_l + 3b(k'_1 + k'_2 - k_1 - k_2) < 3b \ \wedge\ 0 \le j_l + b(k'_1 + k_2 - k_1 - k'_2) < 2b \\ k_1.b \le i_l + 3j_l < (k_1+1).b \ \wedge\ k_2.b \le i_l - 3j_l < (k_2+1).b \\ -b < j_l \le b \ \wedge\ -2b < i_l + j_l \le 2b \ \wedge\ -2b < j_l - i_l \le 2b\end{gathered}$$

□

Extending the results of this section to Presburger sets is now trivial. In order to partition a Presburger set, we consider it as the image of a polyhedron by a projection function. Thus, we can this apply Theorems 1 and 2, respectively, to partition this polyhedron and projection function, and then compute their new image.

4 Implementation and evaluation

We developed two implementations of monoparametric partitioning. One of them is a Java implementation in the *AlphaZ* system [51]. This implementation only covers rectangular tile shapes. The other one is a standalone `C++` implementation[6] of the polyhedron and affine function transformations. We also interfaced the second implementation with a source-to-source `C` compiler, enabling it to generate monoparametric tiled code.

We first report on the scalability of the transformation itself, by studying the Java implementation. Then, we will study the quality of the tiled code generated using the output of the `C` compiler. We ran our experiments on a standard workstation with an Intel Xeon E5-1650 CPU with 12 cores running at 1.6 GHz (max speed at 3.8GHz), and 31GB of memory.

4.1 Scalability of the monoparametric tiling transformation

The AlphaZ system takes, as an input an *Alpha* program, which is a generalization of the SARE representation of Section 2.2. The polyhedron and affine function are directly exposed in the AST, and the program is free of scheduling and memory mapping information. Thus, there is no difference between partitioning tiling of Alpha programs.

We implemented the partitioning transformation is implemented in AlphaZ, as a relatively simple rewrite of the original program, where piecewise affine functions are replaced by case branches. This program is then "flattened out" to bring it back into the form of an SARE, using the normalization transformation. This can produce many polyhedra and branches, and the number of resulting objects is considerable.

We implemented a number of optimizations to improve the scalability. For example, we use the fact that the block and local indices are distinct, and indeed, the polyhedra we manipulate are actually the cross-products of separate block-level and tile-level polyhedra. This reduces the cost of the polyhedral operations we need to perform. We also provide additional options to the monoparametric partitioning transformation that reduce the size of the transformed objects:

- We can force the parameters of the program to be a multiple of the block size parameter (i.e., if N is a parameter, $N = N_b.b$).

[6] Available at `https://github.com/guillaumeiooss/MPP`

- We can specify a minimal value for the block size parameter b. This is especially useful for long, but uniform dependences, which might otherwise access non-neighbor tiles.
- We can specify a minimal value for the block parameters (such as N_b, where N is a parameter).

To study the scalability of our implementation, we measured the time taken by this transformation in our compiler framework, and by an arbitrary polyhedral analysis on the transformed program. We use the Polybench/Alpha[7] benchmark, an hand-written *Alpha* implementation of the *Polybench 4.0* benchmark.

We run the following experiment for each kernel:

- After parsing the program, we apply the rectangular monoparametric partitioning transformation. Because the partitioning transformation is the reindexing part of a tiling, we do not have any legality condition to respect. Thus, we select by default a rectangular tiling of ratio 1^d where d is the number of dimensions of a variable. We assume that the program parameters (N_b) are multiples of the block size parameter (b) and we impose a minimal value for both of them.
- We also apply a representative polyhedral analysis called the *context domain calculation* after monoparametric partitioning. This transformation traverses all the nodes of the program AST, and computes the *context domain* at each one. The context domain of an expression is the set of indices on which the expression value is needed to compute the output of a program. This analysis performs a tree traversal of the AST of the program, and regularly performs polyhedral operations (such as image and preimage) at certain nodes of the AST. It thus, stresses the scalability of polyhedral analysis after monoparametric partitioning.

Figure 7 reports the time taken (in milliseconds) by each phase for all benchmarks of Polybench/Alpha, and also the program after the partitioning transformation.

The time taken by the transformation itself remains reasonable (no more than about 2 seconds for heat-3d). However, the time taken by the subsequent polyhedral analysis (i.e., the context domain calculation) is huge for the stencil kernels (the last six kernels in the bottom table), with heat3d taking up to about 37 minutes). This is due to the size of the program after partitioning and the fact that the context domain analysis builds a polyhedral set per node of the AST.

The main reason a partitioned stencil computation is so big is because of the multiple uniform dependences (of the form $(\vec{i} \mapsto \vec{i} + \vec{c})$ where $\vec{c}$ are constants) in its computation. For each such dependence, the partitioned piecewise affine function has a branch per block of data accessed. Thus the normalized partitioned program will have a branch of computation per combination of block of the data accessed. Even if we progressively eliminate empty polyhedra during normalization, we still have a large number of branches that cannot be merged. Because all the branches contain useful information, we cannot further reduce the size of this program.

4.2 Quality of the monoparametric tiled code

We now consider the source-to-source `C` compiler implementation. Our goal is the compare the quality of a monoparametric tiled code with a fixed-size tiled code. We produce these two codes with the same compiler framework, the only different optimization decision being the nature of the tiling performed.

For each Polybench/C kernel, we use the Pluto compiler to obtain a set of valid tiling hyperplanes. For 5 of these kernels, no legal tiling was found by Pluto, thus we ignore these kernels. We provide manually these tiling hyperplane to our polyhedral compiler in order to replicate the tiling decision. The fixed-size tiled code generated is using tile sizes of 16.

[7]`http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/`

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg	doitgen	mvt	cholesky
Parsing	121	69	62	83	50	118	83	54	43	93	112	51	51	54	55	389
Partitioning	300	157	151	178	93	282	439	119	82	308	482	112	113	187	159	369
Context Domain	1147	504	163	230	162	1257	685	153	207	319	451	153	153	185	201	1197
Num AST Nodes	110	66	21	47	29	136	36	21	25	34	39	25	25	13	29	113
Num Equations	10	6	2	5	3	14	3	2	3	4	6	4	4	2	4	15

Time taken (ms)	durbin	gramschmidt	lu	ludcmp	trisolv	deriche	floyd-warshall	nussinov	adi	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Parsing	121	147	106	179	74	468	220	122	546	331	139	134	183	278
Partitioning	266	398	284	472	139	1213	390	380	2393	1048	678	628	550	3275
Context Domain	2182	1867	1208	2672	203	2843	335	6845	2m 32s	1m 52s	2913	58s	1m 28s	37m 13s
Num AST Nodes	315	123	138	216	39	659	27	537	11931	4194	334	2836	4684	50170
Num Equations	34	20	20	30	5	40	4	57	570	495	38	194	210	1242

Figure 7: For each benchmark in Polybench/Alpha, the first two rows show the time taken (in ms) by the AlphaZ system to perform (i) hyperrectangular monoparametric partitioning transformation, including the normalization step, and (ii) the context domain calculation. The other roes show the program size (as measured by the number of nodes of the AST, and the number of equations) of the partitioned program. All stencil programs in the benchmarks suite (adi to heat-3d on the second row) are first order stencils.

In order to conserve the memory mapping, we apply the monoparametric tiling transformation only on the iteration space. We use the reindexing function to recover the original indexes and use the original array access functions. For example, in the case of a square rectangular tiling of tile size b and an array access `A[i][k]`, we would generate `A[ib*b+il][kb*b+kl]`.

We did not expose parallelism or vectorization opportunities in the generated tiled code. The generated C code was compiled using gcc (version 6.3), with option `-O3` enabled. The problem sizes considered are the ones corresponding to Polybench large dataset, except for the heat-3d kernel. Indeed, for this kernel, the generated code was too large and the compiler ran out of memory. Thus, we have considered instead the nearest power of 2 as problems sizes and we have generated a simplified monoparametric tiled code in which we assume that the tile size parameter divides the problem size parameters.

The execution times[8] are shown in Figure 8. For the majority of the kernels, the execution time of both tiled code are comparable. However, we notice that the monoparametric code is sometimes twice as fast as the fixed-size code. When substituting the tile size parameter with a constant in the monoparametric tiled code, we obtain similar performance. Thus, this is caused by the difference of the structure of the code generated by Cloog. Indeed, the inner loop iterator is not the same: the original iterator is used for the fixed-size tiled code (starting at the origin of the current tile) while the monoparametric code uses i_l (starting at 0). Also, the monoparametric code explicitly separates the tile shapes into different internal loops. This leads to bigger code, but allows the factorization of some terms across loops.

[8]The generated tiled codes are available at `https://guillaume.iooss.fr/CART/TACO_MPP/polyb_exp.tar.gz`

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg
Fixed-size	949	944	776	27.5	3.32	1416	939	617	618	1420	2460	19.2	21.3
Monoparametric	843	945	945	33.2	3.20	1616	928	575	700	1279	2814	17.6	22.7

Time taken (ms)	doitgen	mvt	cholesky	gramschmidt	lu	trisolv	floyd-warshall	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Fixed-size	424	21.0	2054	3093	4255	2.94	20179	2515	5.22	2797	13540	5395
Monoparametric	418	20.8	1108	3107	2357	1.63	19021	1636	7.84	2300	13438	3746

Figure 8: Comparison of the execution time between a fixed-size tiled code and a monoparametric tiled code, given the same compiler framework and optimization parameters. Each number reported is the average of 50 executions.

5 Related work

We already presented some of the fundamental work on tiling [24, 50] in Section 2.3, Characteristics like tile shape, fixed-size vs parametric, legality were already discussed there. In this section, we focus on how tiling is managed in the current polyhedral compilers, specifically in terms of code generation. We will first consider the case of fixed-size tiling, before considering parametric tiling.

5.1 Code generation for fixed-size tiling

Fixed-size tiling is a polyhedral transformation, i.e., the transformed program is still polyhedral. This means that we have two options when applying the fixed-size tiling transformation: either we compute the intermediate representation of the program after transformation, or we generate directly the code using a polyhedral code generator (such as Cloog [9]).

Pluto [10] is a fully automatic source-to-source compiler that generates fixed-size tiled and parallel code. It automatically finds a set of valid tiling hyperplanes by formulating and solving an integer linear programming problem. Because of the problem formulation, the normal vector of hyperplanes are forced to be positive in the original paper, however this limitation was removed in a recent work [1]. After deciding on a set of hyperplanes, Pluto tiles specifically identified bands (i.e., dimensions) of the scheduling functions, and immediately generates the syntax tree of the tiled code using Cloog.

In comparison, our monoparametric tiling transformation explicitly computes the intermediate representation of the tiled program. Because of the size of the resulting program, it might cause some scalability issues for subsequent polyhedral analysis. However, we need to keep all the information about the computation of each tile, thus we do not have a choice. Also, after normalizing and partitioning a program, we obtain a natural classification of the tiles according to their computation. Kong et al. [28] use a similar classification (called *signature*) for their dynamic dataflow compiler framework. However, instead of differentiating each tile according to its computation, they differentiate tiles according to their incoming and outgoing inter-tile dependences.

5.2 Code generation for parametric tiling

Because parametric tiling is a non-polyhedral transformation and prevents any subsequent polyhedral analysis, current compilers integrate this transformation in the code generation phase.

Parametric tiling is simple when the iteration domain is rectangular, the easiest solution is to use a rectangular bounding box of the iteration space and tile it. However, if the iteration domain is, for example, triangular, many of the executed tiles are empty and such a method becomes inefficient.

Renganarayanan et al. [40, 41] presented a parametric tiled code generator for perfectly nested loops and rectangular tiling, which only iterates over the non-empty tiles. The main idea of this approach is to compute the set of non-empty tiles (called *outset*) and the set of full tiles (called *inset*) in a simple way, then use this information to enable efficient code generation. This work was later extended to manage multi-level tiling [27, 41]. We notice that the outset and inset appears in our monoparametric tiling transformation: the outset is the union of the domains on the block indices of all our tiles, and the inset is the union of all the domains on the block indices of only the full-tiles.

Kim [25] proposed another parametric code generator called *D-tiling* for perfectly nested loop, following the work from Renganarayann. Its main insight is the idea that code generation can be done syntactically on each tiled loop incrementally, instead of all at once. It has been extended in order to manage imperfectly nested [26].

Independently, Hartono et al [23] presented a code generation scheme called *PrimeTile* which also manages imperfectly nested loop. The main idea is to cut the computation into stripes, and to place the first tile origin on this stripe at the point where we start to have full tiles in this stripe. The generated code is sequential and efficient [43]. Because the tile origins of different stripes are not aligned, we cannot find a wavefront parallelism and this scheme cannot be adapted to generate parallel tiled code.

Later, Hartono et al [22] presented a code generation scheme called *DynTile* which manages to generate parallel tiled code for imperfect nested loop. The idea is to consider the convex hull of all statements, then to rely on a dynamic inspector to determine the wavefronts of tiles, which are scheduled in parallel. Finally, Baskaran et al [7] have presented *PTile* which allows parametrized parallel tiled code for imperfectly nested affine loops. This algorithm is identical to the one used in D-tiler, and was independently developed. A survey [43] compares the effectiveness of the sequential, and the parallel code generated by Primetile, Dyntile and PTile.

Another approach is to adapt the Fourier-Motzkin elimination procedure to manage parametric coefficients. This has been done by Amarasinghe [5] who integrated the possibility of managing linear combinations of parametric coefficients in the SUIF tool set (such as $(N + 2M).i$, where N and M are parameters, and i is a variable), but no details have been provided and only perfectly nested loops were managed. Lakshminarayanan et al [41] (Appendix B) extended this to the case where the coefficients of a linear inequality can be parameters.

More generally, several authors have looked at extending the polyhedral model to be able to manage parametric tiling naturally. GrÃ¶sslinger et al [21] extended the polyhedral model to deal with parametrized coefficients, and showed how to adopt Fourier-Motzkin and the simplex algorithm. In particular, these coefficients can be rational fractions of polynomials of parameters. However, they have to rely on quantifier elimination, thus their method has scaling issues. Achtziger et al [2] studied how to find a valid quadratic schedules for an affine recurrence equation. Recently, Feautrier [17] considered polynomial constraints and has presented an extension of the Farkas lemma. This class encompasses the parametric tiling transformation, at the cost of the complexity of the analysis.

6 Conclusion

We presented the monoparametric tiling transformation, a polyhedral tiling transformation which allows for partial parametrization. We have decomposed this transformation into two components: the partitioning transformation which is the reindexing transformation which introduces the new tiling indices, and the tiling transformation which is changing the schedule of the program to satisfy the atomicity property of a tile. We have shown that the monoparametric partitioning transformation is a polyhedral transformation and transform a polyhedron into a union of Presburger sets and an affine function into a piecewise affine function with modulo conditions. These closure properties works for any polyhedral tile shapes. Finally, we have evaluated the scalability of this transformation and we have discovered some issues with the size of the partitioned program for stencils computations.

The work presented in this paper is the main basic block of the monoparametric tiling transformation. The major advantage of this transformation is its flexibility of usage inside a compiler flow. First, we can support any shape of tile and produce a parametrized code, which extends the prior work on tiled code generation for some tile shapes (such as hexagonal tiling). It also means that any future shape found to be interesting for tiling will be able to have a monoparametric tiled code generator at a low implementation cost. Moreover, because the resulting program is polyhedral, we can still use the polyhedral analysis and transformation after the tiling transformation, whereas in a classical parametric tiling code generator, this transformation has to be embedded in the code generation phase. In particular, we can reapply multiple times the monoparametric tiling transformation on the transformed program, in order to produce multiple level of tiling.

Our main claim is that monoparametric partitioning/tiling allows the benefits of remaining in the polyhedral model, while still providing a limited form of parameterization. Today, there are relatively few tools that perform any post-tiling analyses or optimizations. One example is the work of Kong et al. [28] who use a (fixed-size) tiled program to generate codes for a data-flow language. We believe that this is a chicken and egg issue: our work will spur research in such directions.

The monoparametric partitioning applications to polyhedra and affine functions are implemented as a C++ standalone library. A version of this work restricted to rectangular tile has been integrated as a program transformation of the AlphaZ system. We have interface the C++ standalone library with a source-to-source C compiler and compare the quality of the fixed-sized tiled and the monoparametric tiled code generated, under a common compiler code generator.

Références

[1] Aravind Acharya and Uday Bondhugula. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. *SIGPLAN Notices*, 50(8):54–64, January 2015.

[2] Wolfgang Achtziger and Karl-Heinz Zimmermann. Finding quadratic schedules for affine recurrence equations via nonsmooth optimization. *Journal of VLSI Signal Processing Systems*, 25(3):235–260, July 2000.

[3] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, 2007.

[4] Christophe Alias and Alexandru Plesco. Data-aware Process Networks. Research Report RR-8735, Inria - Research Centre Grenoble – Rhône-Alpes, June 2015.

[5] Saman Prabhath Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*. PhD thesis, Stanford University, 1997.

[6] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 1–11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[7] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th Annual IEEE/ACM*

International Symposium on Code Generation and Optimization, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[8] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[9] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[11] Jichun Bu, Ed F Deprettere, and P Dewilde. A design methodology for fixed-size systolic arrays. In *Application Specific Array Processors, 1990. Proceedings of the International Conference on*, pages 591–602. IEEE, 1990.

[12] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *INTEGRATION, the VLSI journal*, 12(3):293–304, 1991.

[13] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 298–308, New York, NY, USA, 2003. ACM.

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

[15] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[16] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[17] Paul Feautrier. The power of polynomials. In Alexandra Jimborean and Alain Darte, editors, *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, pages 1–5, Amsterdam, Netherlands, January 2015.

[18] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[19] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66–75, New York, NY, USA, 2014. ACM.

[20] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly – polyhedral optimization in LLVM. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 1–6, Chamonix, France, 2011.

[21] Armin Grosslinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. Technical report, Technische UniversitÃ¤t MÃ¼nchen, 2004.

[22] A. Hartono, M.M. Baskaran, J. Ramanujam, and P. Sadayappan. Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In *International Symposium on Parallel Distributed Processing (IPDPS),*, pages 1–12, April 2010.

[23] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[24] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'88, pages 319–329, January 1988.

[25] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] DaeGon Kim and Sanjay V. Rajopadhye. Parameterized tiling for imperfectly nested loops. Technical Report CS-09-101, Colorado State University, February 2009.

[27] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay V. Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, page 51, 2007.

[28] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Transactions on Architecture and Code Optimization*, 11(4):61:1–61:30, January 2015.

[29] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN conference of Programing Language Design and Implementation*, 42(6):235–244, June 2007.

[30] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 63–74. ACM, 1991.

[31] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3(3):173–182, September 1991.

[32] C. Mauras. *ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[33] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Geogres-André Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proceedings of the 4th GCC Developper's Summit*, pages 1–18, Ottawa, Ontario, Unknown or Invalid Region, 2006.

[34] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *The International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

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

[36] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, October 2000.

[37] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 1(2):95–113, 1989.

[38] Sanjay V Rajopadhye, S Purushothaman, and Richard M Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 488–503. Springer, 1986.

[39] D. A. Reed, L. M. Adams, and M. L. Partick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, 36(7):845–858, July 1987.

[40] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 405–414, June 2007.

[41] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized loop tiling. *ACM Trans. Program. Lang. Syst.*, 34(1):3, 2012.

[42] Robert Schreiber and Jack J Dongarra. Automatic blocking of nested loops. 1990.

[43] Sanket Tavarageri, Albert Hartono, Muthu Baskaran, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. Parametric tiling of affine loop nests. In *15th Workshop on Compilers for Parallel Computing (CPC'10)*, pages 1–15, Vienna, Austria, July 2010.

[44] Jürgen Teich and Lothar Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14(3):297–332, 1993.

[45] Konrad Trifunovic and Albert Cohen. Enabling more optimizations in GRAPHITE: ignoring memory-based dependences. In *Proceedings of the 8th GCC Developper's Summit*, Ottawa, Canada, October 2010.

[46] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software (ICMS'10)*, LNCS 6327, pages 299–302. Springer-Verlag, 2010.

[47] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998.

[48] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[49] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[50] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[51] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay V. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012*, pages 17–31, September 2012.

Table des matières

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

ISSN 0249-6399