Monoparametric Tiling of Polyhedral Programs

Guillaume Iooss, Christophe Alias, Sanjay Rajopadhye

To cite this version:

Guillaume Iooss, Christophe Alias, Sanjay Rajopadhye. Monoparametric Tiling of Polyhedral Programs. 2021. hal-02493164v2

HAL Id: hal-02493164

https://hal.inria.fr/hal-02493164v2

Preprint submitted on 3 Jan 2021

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

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

Monoparametric Tiling of Polyhedral Programs

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

February 27, 2020

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 with modulo conditions, and (ii) to an affine function produces a piecewise-affine function with modulo conditions. 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.

Keywords: Tiling, Program Transformation, Polyhedral Model, Compilation

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* [41, 39, 14, 15, 16] has emerged over the past few decades. A polyhedral program is a program where "iteration spaces" are unions 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. Many important categories of program are polyhedral, including dense linear algebra algorithms, convolution, stencils and some graph-theory computations.

[*]Ecole Normale Supérieure. guillaume.iooss@gmail.com
[†]ENS-Lyon/Inria. Christophe.Alias@inria.fr
[‡]Colorado State University. sanjay.rajopadhye@colostate.edu

```
for (i=0; i<N; i++)
  for (j=0; j<i; j++)
    ... S[i][j] ...
```

(a) Original code

```
for (ib=0; ib < ceil(N/32); ib++)
  for (jb=0; jb <= ib; jb++)
    for (i=32*ib;
         i<min(32*ib+32, N); i++)
      for (j=32*jb;
           j<min(32*jb+32, i); j++)
        ... S[i][j] ...
```

(b) Fixed-size tiling (tile sizes: 32×32)

```
for (ib=0; ib < ceil(N/b1); ib++)
  for (jb=0;
       jb <= floor((b1*(ib+1)-1)/b2);
       jb++)
    for (i=b1*ib;
         i<min(b1*(ib+1), N); i++)
      for (j=b2*jb;
           j<min(b2*(jb+1), i); j++)
        ... S[i][j] ...
```

(c) Parametric tiling (tile sizes: $b1 \times b2$)

Figure 1: Example of fixed-size and parametric tiling. The parametric tiling code has non-affine constraints on the innermost loop bounds.

Among these program transformations, iteration space tiling [24, 53, 30] (also called loop blocking [45] or partitioning [11, 12, 48]) 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 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 (cf example in Figure 1b : the boundary conditions of the innermost loop are still affine and we now have four loops instead of two). 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 autotuning step [18, 36, 51]) without having to recompile. However, the program is no longer polyhedral after transformation, thus no subsequent polyhedral analysis or transformation can be applied (cf example in Figure 1c : the boundary conditions of the innermost loop are quadratic and not affine). 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, 44] 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 (with a fixed ratio of 2×1). However, a $b1 \times b2$ tiling (with $b1$ and $b2$ two program

parameters) is not a monoparametric tiling (and does not have a fixed ratio).

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 prove that the monoparametric tiling transformation is a polyhedral transformation. To do so, we decompose the tiling transformation into two parts: the *partitioning*, which is a reindexing of the original program (introducing the new indices, without changing the order of execution), and the actual *tiling*. We show that the reindexing part of the transformation is a polyhedral transformation, by proving that:
  - Applying the monoparametric partitioning to a polyhedron returns a set which can be expressed as a finite union of $\mathbb{Z}$-polyhedra (i.e., polyhedra with modulo conditions).
  - Applying the monoparametrically partitioning the input and the output spaces of an affine function transforms it into a piecewise affine function with modulo constraints.
- Using these two main results, we show how to apply the monoparametric tiling transformation to a polyhedral program, and we study its influence on the compilation time. We support any polyhedral tile shape which tessellates the space, including those whose tile origin might not always be integral.
- We implemented the monoparametric partitioning transformation on polyhedra and affine functions as both a standalone tool[1] written in C++, and also in the *AlphaZ* polyhedral compiler [55]. We also integrated the standalone library inside a polyhedral source-to-source `C` compiler[2].
- We compare the efficiency of monoparametrically tiled code with the corresponding fixed-size code, both produced by the same polyhedral compiler and with similar transformation parameters. We show that the monoparametric tiled code performs similarly compared to the fixed-size tiled code, while being a strictly more general code due to its parametrization.

We start in Section 2 by introducing the background 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 our implementations of monoparametric partitioning of polyhedral programs, and also compare monoparametrically tiled code with fixed-size tiled code. We describe related work in Section 5, before concluding in Section 6.

2 Background: polyhedral compilation and tiling

The *polyhedral model* [41, 39, 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, 37] or code generation [38, 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 [52, 10] or memory

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

[2] An online demonstration is available at `http://foobar.ens-lyon.fr/mppcodegen/index.php`

footprint [13] to name a few. The input language is usually imperative (C like) [10, 4]. It may also be an equational language [33, 55]. Today, polyhedral optimizations can also be found in the heart of production and research compilers [35, 49, 19, 3].

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 [41, 33] 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.

2.1 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 integers 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 between an affine expression and zero (or equivalently, between two affine expressions).

We focus on two objects: polyhedra and affine functions. A *polyhedron* [46] is a set of 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 building blocks of the *polyhedral model*. A polyhedron is used to represent the set of instances of a statement in 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 involve *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.*

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:

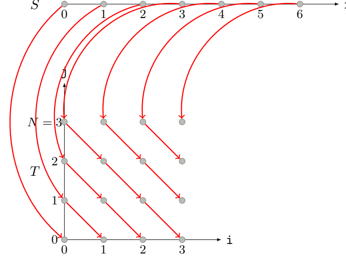
$$\mathcal{P} = \left\{ i, j \;\middle|\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} . \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} .(N) + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$$

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 matrices, $\vec{c}$ an vector and $\vec{p}$ are the program parameters.*

for i := 0 **to** $2N$
S: c[i] := 0;

for i := 0 **to** N
for j := 0 **to** N
T: c[i+j] := c[i+j] + a[i]*b[j];

(a) Product of polynomials



(b) Iteration domains and producer/consumer dependences

$$S[i] = \; 0 \leq i \leq 2N : 0$$

$$T[i,j] = \begin{cases} i = 0 \vee j = N : \\ \quad S[i+j] + a[i] * b[j] \\ 2 \leq i \leq N \wedge 1 \leq j \leq N-1 : \\ \quad T[i-1, j+1] + a[i] * b[j] \end{cases}$$

(c) SARE

Figure 2: System of affine recurrence equation of the product of polynomial. The left part shows the associated tiled loop, the middle a graphical representation of its iteration domains and the right part the corresponding SARE.

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.

A $\mathbb{Z}$-polyhedron [34, 40] is defined as the intersection of a polyhedron and an integral lattice. While a d-dimensional polyhedron is a subset of $\mathbb{Q}^d$, whereas a $\mathbb{Z}$-polyhedron is a subset of $\mathbb{Z}^d$. We will call *integral polyhedron* a $\mathbb{Z}$-polyhedron whose lattice is the canonic lattice $\mathbb{Z}^d$, and to emphasize the distinction, we will call a classical polyhedron a *rational polyhedron*. Polyhedral compilation uses integral polyhedra, since they represent a discrete collection of points (such as the iteration domain). This allows us more options, such as transforming a strict inequality constraint of an integral polyhedron into an inequality constraint. In the rest of the paper, unless explicitly stated, we use integral polyhedra. Likewise, affine functions only have integral coefficients.

In general, the lattice of a $\mathbb{Z}$-polyhedron may be non-canonic, which corresponds to modulo conditions (of the form $i \bmod n = c$, for some constants c and n). Most modern libraries [32] support such sets, or a generalization (e.g., the Integer Set Library [50] even supports Presburger sets, i.e., union of integral polyhedra with existential indices).

In Section 3, we present our main results about closure properties for integral polyhedra and affine functions.

2.2 Program Representation

In this paper, we use a program representation called *System of Affine Recurrence Equations* (SAREs). A computation is represented by a set of equations of the form:

$$\texttt{Var}[\vec{i}] \; = \; \begin{cases} \dots \\ \vec{i} \in \mathcal{D}_k : \texttt{Expr}_k \\ \dots \end{cases}$$

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 `Var[f(ĩ)]` where `f` is an affine function

– A constant `Const`,

– An operation $\texttt{Op}(\ldots \texttt{Var}_1[\texttt{f}_\texttt{i}[(\tilde{\texttt{i}})], \ldots)]$ of arity k (i.e., there are k terms of the form $\texttt{Var}_\texttt{i}[\texttt{f}_\texttt{i}(\tilde{\texttt{i}})]$)

In a slightly more general form of SARE, we allow the case branches and restrictions to be arbitrarily nested, and furthermore, allow arbitrary expressions as the arguments to an `Op` by modifying the last clause to $\texttt{Op}(\ldots, \texttt{Expr}_\texttt{i}, \ldots)]$. This is the representation used in the Alpha language [33, 31]. Note that this Alpha representation, while seemingly more general than SAREs, is mathematically equivalent. Indeed, Mauras [33] proposed a *normalization* procedure (which is implemented as a program transformation in the AlphaZ system [55]) that systematically "flattens out" any Alpha program into the "normal" SARE form.

We have a *dataflow dependence* between these two equations when the computation of an equation uses a data produced by another equation. This implies a precedence constraint on the order of execution, called *schedule*: the computation of the producer equation must happen before the computation of the consumer. An SARE only has the dataflow dependences, and does not have any information about scheduling and memory allocation (i.e., information about how a data is stored, different data could be sharing the same memory location, introducing additional kind of dependences).

Moreover it can be provided with equational (denotational) semantics, allowing us to develop and reason about semantics preserving transformations of SAREs, independent of this information. Our results can be carried over to any other polyhedral representation, and we demonstrate this in Section 4, by integrating them into a C compiler that uses a loop-based representation.[3] Fig. 2.(a) shows a simple program fragment that computes the product of two polynomials. Fig. 2.(b) and (c) show, respectively, the producer/consumer dependences between loop iterations, and the SARE, which is nothing more than a compact representation of the former.

2.3 Tiling: the essential transformation

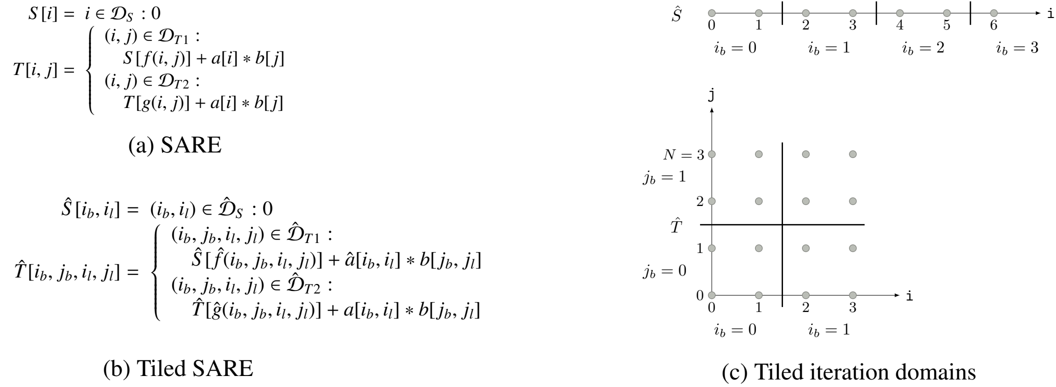
Tiling [24, 54] is a program transformation that partitions an iteration set into subsets called *tiles*, and schedules each tiles atomically: we finish the computation of a tile before starting on a new one. This implies that cyclic dependences between tiles are prohibited.

One important aspect of tiling is *tile shape*. The most commonly used shape is a hyper-parallelepiped, defined by its boundary hyperplanes [24, 54]. A particular case is *orthogonal tiling* (also called *rectangular 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 [42, 20]. 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 rectangular 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.

We consider the tiling transformation as a two-part transformation: we call the first part *partitioning*, which is a *reindexation* of the original domains and dependences by introducing new dimensions. From the original dimensions $\vec{i}$, it introduces new dimensions to identify a tile (*block indices*, $\vec{i_b}$) and a point inside a tile (*local indices*, $\vec{i_l}$), thereby doubling the number of dimensions (assuming that tiles partition along all dimensions). This

[3] See the online demonstration at `http://foobar.ens-lyon.fr/mppcodegen/index.php`.



(a) SARE

(b) Tiled SARE

(c) Tiled iteration domains

Figure 3: Application of the tiling transformation to the product of polynomial example presented in Figure 2

part of the transformation is always legal, and does not impact the schedule. The second part is a modification of the schedule by using these newly introduced dimensions while ensuring atomicity.

For example, we consider a perfectly nested loop program that we wish to tile using a rectangular tile shape. The partitioning part of the tiling corresponds to a succession of loop strip-mining transformation applied separately to all the dimensions. The second part is a succession of loop interchange, which push the loops concerning the local indices on the innermost dimensions. Notice that for more complex tile shapes, we cannot describe the impact on the program by a succession of simpler program transformations.

When the program representation is in the form of an SARE (i.e., without any explicit schedule) the tiling transformation is just a partitioning—also called a reindexing. A tiling transformation $\mathcal{T}_S$ maps each iteration $\vec{i} \in \mathbb{Z}^n$ of the statement S to a *tiled iteration* $(\vec{i_b}, \vec{i_l}) \in \mathbb{Z}^{2n}$, where $\vec{i_b}$ denotes the block indices and $\vec{i_l}$ denotes the local indices. Given a tiling transformation $\mathcal{T}_S$ for each SARE array S, *tiling an SARE* means:

- Reindexing arrays with block and local indices $S[\vec{i}] \mapsto \hat{S}[\vec{i_b}, \vec{i_l}]$.
- Tiling each equation domain of $\hat{S}$ with $\mathcal{T}_S \colon \mathcal{D} \mapsto \hat{\mathcal{D}}$.
- Rewriting array index functions with block and local indices: $f \mapsto \hat{f}$.

On our motivating example, a possible tiling is given by

$$\mathcal{T}_S(i) = (\lfloor i/2 \rfloor, i \bmod 2) \qquad \text{and} \qquad \mathcal{T}_T(i, j) = (\lfloor i/2 \rfloor, \lfloor j/2 \rfloor, i \bmod 2, j \bmod 2)$$

Figure 3 (a, b) illustrates this transformation on the SARE (index names are retained as symbolic), and (c) shows the iteration domains of S and T after tiling. Tiled iteration domains are obtained by writing the euclidian division:

$$\hat{\mathcal{D}}_S = \{(i_b, i_l) \mid 0 \le \mathcal{T}_S^{-1}(i_b, i_l) < 2N \ \wedge \ 2i_b \le \mathcal{T}_S^{-1}(i_b, i_l) \le 2(i_b + 1)\}$$

Since $\mathcal{T}_S^{-1}(i_b, i_l) = 2i_b + i_l$ is affine, $\hat{\mathcal{D}}$ is a polyhedron. Hence it satisfies the constraints of the SARE representation. This closure property is unclear when the tile size b is *parametric*, since $\mathcal{T}_S^{-1}(i_b, i_l) = b.i_b + i_l$ is then a quadratic form.

Index functions are renamed to account for tiled indices. The source and the target domains might be different, with different dimensions and partitionings (e.g., with $f : \mathbb{Z}^2 \to \mathbb{Z}$), which makes it challenging to determine them

automatically. For instance, in the original SARE, the index function g is defined by: $g(i, j) = (i-1, j+1)$. In the tiled SARE, we substitute g with its tiled counterpart, $\hat{g} = \mathcal{T}_S \circ g \circ \mathcal{T}_T^{-1}$. Tiled index functions are piecewise: when $(i_l - 1, j_l + 1)$ is a valid local index, $\hat{g}(i_b, j_b, i_l, j_l) = (i_b, j_b, i_l - 1, j_l + 1)$. On the corner case, we take the value from a neighbor tile. Finally, we end-up with a piecewise affine definition of $\hat{g}$, which fits the SARE constraints. Again, the closure is unclear with parametric tile size $\vec{b}$, as a direct application of $\hat{g} = \mathcal{T}_S \circ g \circ \mathcal{T}_T^{-1}$ would lead a non-affine function.

Once the partitioning transformation is applied, the reindexed SARE can be scheduled by a polyhedral backend to produce a tiled sequential program: we simply provide the backend with a schedule and a memory allocation function for each array. The derivation of a valid tiled schedule is completely orthogonal to our paper, and any state of the art scheduler may be used. In our example, one possible schedule is $\theta_T(i_b, i_l, j_b, j_l) = (i_b, j_b, i_l, j_l)$.

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 $\mathbb{Z}$-polyhedra and correspondingly, monoparametric partitioning of an affine function yields a piecewise affine function with modulo conditions. 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:*

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

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, $\mathcal{P}$, by a constant a, is the set denoted by $a \times \mathcal{P}$, and defined by $a \times \mathcal{P} = \{\vec{z} \mid (\vec{z}/a) \in \mathcal{P}\}$. Using this notion, we define a monoparametric partitioning as simply the homothetic scaling of a fixed-size partitioning by a parametric factor:

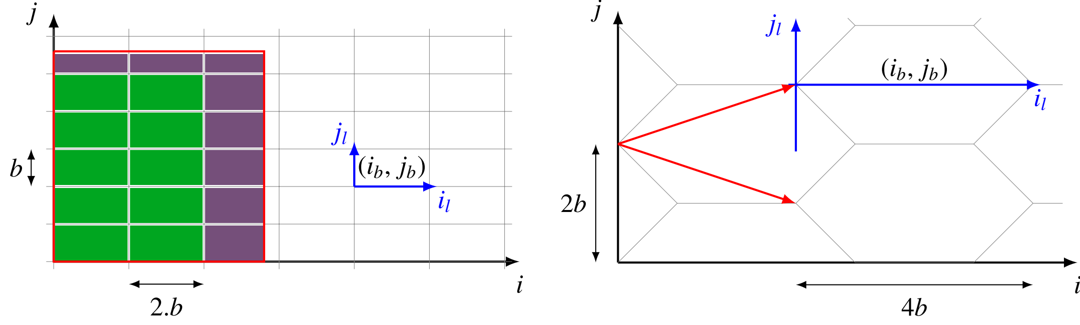


Figure 4: On the left: example of rectangular monoparametric partitioning for the array C of the matrix multiplication. On the right: example of hexagonal monoparametric partitioning for a 2D space. (i_b, j_b) are the block indices, which identify a tile. (i_l, j_l) are the local indices, which identify 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.

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:*

- *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 $\vec{i_l} \in \mathcal{P}_b$

Example 1. *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).*

Example 2. *Figure 4 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 $\mathcal{P}$ means computing the image $\mathcal{T}_b(\mathcal{P})$ of $\mathcal{P}$ 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 a reindexing: 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 $\mathcal{P}$ 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 $\Delta = \mathcal{T}_b(\mathcal{P})$. If we use the lexicographic order 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.

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 $\mathbb{Z}$-polyhedra. 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 $\mathcal{P} = \{\vec{i} \mid Q.\vec{i} + R.\vec{p} + \vec{q} \geq \vec{0}\}$ 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: $\mathcal{T}_b(\mathcal{P})$ of $\mathcal{P}$ by $\mathcal{T}_b$ is given by*

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

where:

- N_{constr} *is the number of constraints of the polyhedron* $\mathcal{P}$.
- $\vec{p} = \vec{p_b}.b + \vec{p_l}$ *with* $\vec{0} \leq \vec{p_l} < b.\vec{1}$. $\vec{p_b}$ *are the tiled parameters and* $\vec{p_l}$ *the local parameters.*
- $X_{c,\bullet}$ *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. When* $D \neq Id$, *this allows us to represent non-integral tile origins, and* $\vec{\alpha}$ *represents the rational shift of the tile origin compared to its integral start.*

- *δ is the least common multiplier of the diagonal elements of D.*
- *k_c^{min} and k_c^{max} are constants, depending on the coefficients of the c-th constraint and the tiling chosen. The value of k_c represents the way the c-th constraint of the polyhedron cuts a tile: $k_c = k_c^{min}$ represents no cut on the tile, and $k_c = k_c^{max}$ is the cut where the most area of the tile is excluded.*

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

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

$\mathcal{P}$ 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. Intuitively, this integral division corresponds to the non-integral tile origin management: every kind of non-integral tile origin is associated to a different $\vec{\alpha}$ ($\vec{\alpha} = \vec{0}$ corresponds to an integral tile origin), and the integral division allows us to differentiate these situations.

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 f_c by constants. Therefore, f_c can only take a constant non parametric number of values, and we have the possibility to create a union of $\mathbb{Z}$-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: $k_c \in [|k_c^{\text{min}}; k_c^{\text{max}}|]$ [4]. Eqn (4) becomes:

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

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

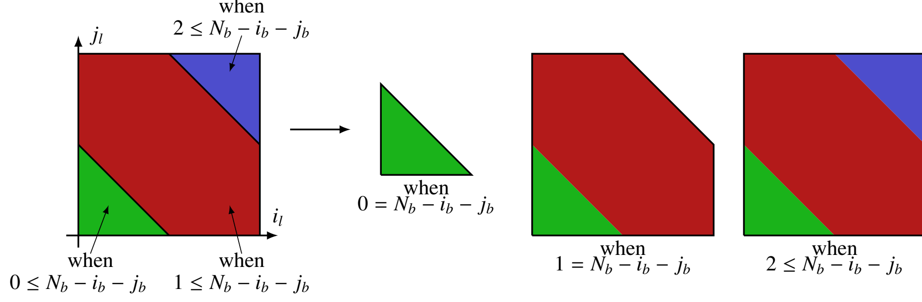


Figure 5: Stripe coverage of a tile. Given a constraint (e.g., $0 \leq N - i - j$), we have obtained a disjoint union of $\mathbb{Z}$-polyhedra, each set covering a stripe of a given tile (as shown on the left part of the figure). By examining the constraints on the block indices (e.g., $0 \leq N_b - i_b - j_b + k_c$), 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.

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 cth constraint of (1) corresponds to the union of sets obtained for each value of k_c:

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

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: Reordering 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 5):

- 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 $\mathcal{T}_b(\mathcal{P})$.

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 $\mathcal{T}_b(\mathcal{P})$.

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

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

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

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

(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 it 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 least common multiple of the diagonal elements of D, such that $\delta.D^{-1}$ is an integral matrix. We get:

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

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

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

□

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 $\mathbb{Z}$-polyhedra 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* $\mathcal{P} = \{\vec{i} \mid Q.\vec{i} + R.\vec{p} + \vec{q} \ge \vec{0}\}$ *with size parameters* $\vec{p}$ *, and a rectangular*

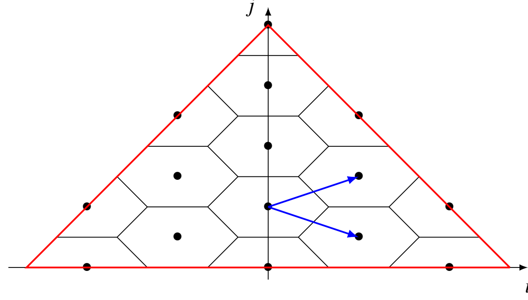


Figure 6: Polyhedron and tiling of Example 3. The dots correspond to the tile origins of the tiles contributing to the polyhedron. The blue arrows show the basis of the lattice of tile origins.

monoparametric partitioning $\mathcal{T}_b$ with shape $\mathcal{P}_b$, tile origin lattice $\mathcal{L}_b$ and size b, then:

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

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. □

Example 3. *Consider the following polyhedron: $\{i, j \mid j - i \leq N \wedge i + j \leq N \wedge 0 < j\}$ and the following hexagonal partitioning:*

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

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 6.

Let us start with the first constraint of the polyhedron.

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

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.

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

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 $2 \times 2 \times 2 = 8$ $\mathbb{Z}$*-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:*

$$(\vec{i'_b}, \vec{i'_l}) = \left(\begin{array}{c} D'.L'^{-1}.Q.L.D^{-1}.\vec{i_b} + D'.L'^{-1}.R.\vec{p_b} + \vec{\alpha'} - D'.L'^{-1}.Q.L.D^{-1}.\vec{\alpha} + D'.L'^{-1}.(\vec{k} - \vec{k'}) \\ Q.\vec{i_l} + R.\vec{p_l} + b.(Q.L.D^{-1}.\vec{\alpha} - L'.D'^{-1}.\vec{\alpha'} + \vec{k'} - \vec{k}) + \vec{q} \end{array} \right)$$

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

where:

- $\vec{p} = \vec{p_b}.b + \vec{p_l}$ *with* $\vec{0} \leq \vec{p_l} < b.\vec{1}$. $\vec{p_b}$ *are the tiled parameters and* $\vec{p_l}$ *the local parameters.*
- $\mathcal{L}_b = b.L.D^{-1}.\mathbb{Z}$ *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. $\vec{\alpha}$ (resp. $\vec{\alpha'}$) represents the rational shift of the tile origin of the input space (res. of the output space) compared to its integral start.*

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 $\vec{p}$:

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

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 least 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:

$$\begin{cases} \vec{i'_b} = \epsilon'.\vec{\lambda'} + \vec{\alpha'}, & \vec{0} \le \vec{\alpha'} < \epsilon'.\vec{1} \\ \vec{i_b} = \epsilon.\vec{\lambda} + \vec{\alpha}, & \vec{0} \le \vec{\alpha} < \epsilon.\vec{1} \end{cases}$$

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

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

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 $[a \ge 0 \Rightarrow \lfloor a \rfloor \ge 0]$).

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

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 $\vec{0} \le \vec{p_l} < b.\vec{1}$, thus $\vec{0} \le \frac{\vec{p_l}}{b} < \vec{1}$. 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:

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

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

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

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. □

Example 4. *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 $\mathcal{T}_b = \{i, j \mid -b < j \le b \ \wedge\ -2b < i + j \le 2b \ \wedge\ -2b < j - i \le 2b\}$ 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 $\mathcal{T}'_b = \{i, j \mid 0 \le i < 3b \wedge 0 \le j < 2b\}$ and lattice $\mathcal{L}'_b = L'.b.\mathbb{Z}^2$ where $L' = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}$*

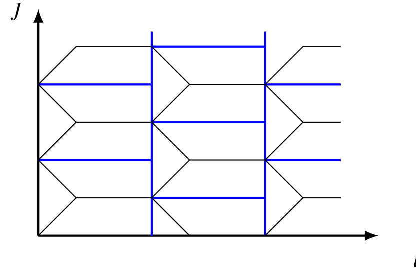


Figure 7: Overlapping of the rectangular (in blue) and the hexagonal tiles in Example 4

An overlapping of these two tilings is shown in Figure 7. The derivation goes as follow:

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

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:*

$$\left(i_b + k_1 - k'_1,\ j_b + k_2 - k'_2,\ \ i_l + 3b(k'_1 + k'_2 - k_1 - k_2),\ j_l + b(k'_1 + k_2 - k_1 - k'_2)\right)$$
$$\textit{when } 0 \leq i_l + 3b(k'_1 + k'_2 - k_1 - k_2) < 3b \ \wedge\ 0 \leq j_l + b(k'_1 + k_2 - k_1 - k'_2) < 2b$$
$$k_1.b \leq i_l + 3j_l < (k_1+1).b \ \wedge\ k_2.b \leq i_l - 3j_l < (k_2+1).b$$
$$-b < j_l \leq b \ \wedge\ -2b < i_l + j_l \leq 2b \ \wedge\ -2b < j_l - i_l \leq 2b$$

Extending the results of this section to a $\mathbb{Z}$-polyhedron is possible. We introduce an additional dimension to express the modulo condition as a linear equality. The original $\mathbb{Z}$-polyhedron is the image of this polyhedron by a projection function which removes these new dimensions (also called existential dimensions). Thus, we can this apply Theorems 1 and 2, respectively, to partition the extended polyhedron and its projection function. Then, we compute the image of the partitioned extended polyhedron by the partitioned projection function.

Time taken (ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg	doitgen	mvt	cholesky	durbin
Num Equations	10	6	2	5	3	14	3	2	3	4	6	4	4	2	4	15	34
Num AST Nodes	110	66	21	47	29	136	36	21	25	34	39	25	25	13	29	113	315
Parsing	121	69	62	83	50	118	83	54	43	93	112	51	51	54	55	389	121
Partitioning	300	157	151	178	93	282	439	119	82	308	482	112	113	187	159	369	266
Context Domain	1147	504	163	230	162	1257	685	153	207	319	451	153	153	185	201	1197	2182

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

Figure 8: For each benchmark in Polybench/Alpha, the first two rows show the size of the partitioned program (as measured by first the number of equations, and then, the number of AST nodes). The other rows give the time (in ms) that the AlphaZ system takes to (i) parse the program, (ii) perform hyperrectangular monoparametric partitioning transformation followed by normalization, and (ii) the context domain calculation. All stencil programs in the suite (adi to heat-3d on the second row) are first order stencils.

4 Implementation and evaluation

We developed two implementations of monoparametric partitioning. One of them is a Java implementation in the *AlphaZ* system [55]. This implementation uses a SARE-like program representation and only covers rectangular tile shapes and parallelogram shapes which can be made rectangular with a change of basis transformation. The other one is a standalone `C++` implementation 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 for a C-like program representation[5].

In this section, we use the Java implementation to report on the scalability of the transformation itself: we evaluate the size of the program representation after the monoparametric tiling transformation and shows how the compilation time is linked to the size of the AST. Then, we use our `C` compiler implementation to compare the quality of the monoparametric tiled code and the fixed-size tiled code, under the same compiler framework and hypothesis. We show that the execution time of both versions are similar. 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

[5]The `C++` library is available at `https://github.com/guillaumeiooss/MPP`, and a demonstration of the compiler is at `http://foobar.ens-lyon.fr/mppcodegen/index.php`

scheduling and memory mapping information.

We implemented the partitioning transformation in AlphaZ, as a relatively simple syntactic rewrite of the original program, where piecewise affine functions are replaced by case branches. This program is then "flattened out" by the normalization transformation mentioned in Sec. 2 to bring it back into the form of an SARE. Normalization traverses the program AST, and makes calling polyhedral operations at each node. This may produce many polyhedra and branches, and the number of resulting objects is considerable. Thus, it serves as an excellent use case to rest the scalability of the partitioning transformation.

We developed 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 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 some additional options to the monoparametric partitioning transformation that reduce the size of the transformed objects:

- The user may optionally force the parameters of the program to be a multiple of the block size parameter (i.e., if N is a parameter, $N = N_b.b$). This option allows us to remove many corner cases.
- 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 normalization transformation in our compiler framework, and also by another polyhedral analysis on the transformed program. We applied these to programs in Polybench/Alpha[6], an implementation of the *Polybench 4.0* benchmark written in the *Alpha* language. We ran the following experiment for each program:

- 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 (a subset of the indices where it is defined) at which the expression value needs to be evaluated, in order 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) when visiting nodes of the AST. Thus, like normalization, it too stresses the scalability of polyhedral analysis after monoparametric partitioning.

Figure 8 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 heat-3d 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.

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

Time taken (in ms)	correlation	covariance	gemm	gemver	gesummv	symm	syr2k	syrk	trmm	2mm	3mm	atax	bicg
Tiling Fixed-size	200	100	100	100	100	100	100	100	100	100	100	100	100
CodeGen Fixed-size	506	203	560	740	730	205	820	720	920	410	1699	100	120
Tiling Monoparametric	328	163	660	370	380	114	570	550	590	213	706	550	310
CodeGen Monoparametric	3530	1920	696	180	142	1177	390	405	491	13770	72434	1400	606
Num of generated sets	47	28	12	14	14	24	12	12	12	36	78	18	12
Exec Fixed-size	949	944	776	27.5	3.32	1416	939	617	618	1420	2460	19.2	21.3
Exec Monoparametric	843	945	945	33.2	3.20	1616	928	575	700	1279	2814	17.6	22.7

Time taken (in ms)	doitgen	mvt	cholesky	gramschmidt	lu	trisolv	floyd-warshall	fdtd-2d	jacobi-1d	jacobi-2d	seidel-2d	heat-3d
Tiling Fixed-size	100	100	100	100	100	100	100	100	100	100	100	100
CodeGen Fixed-size	171	360	121	233	930	660	380	278	530	143	740	216
Tiling Monoparametric	416	210	980	168	700	210	620	1454	146	4517	1516	35775
CodeGen Monoparametric	3687	650	289	2384	420	810	194	11848	2989	33874	6874	38670
Num of generated sets	32	8	18	32	20	8	8	160	42	142	74	140
Exec Fixed-size	424	21.0	2054	3093	4255	2.94	20179	2515	5.22	2797	13540	5395
Exec Monoparametric	418	20.8	1108	3107	2357	1.63	19021	1636	7.84	2300	13438	3746

Figure 9: Comparison of the compile time (of the tiling and code generation passes) and the execution time between a fixed-size tiled code and a monoparametric tiled code, given the same compiler framework and optimization parameters. Each execution time reported is the average of 50 executions, to smooth potential execution time instabilities. We also provide the sum of the number of generated sets produced while applying the monoparametric partitioning transformation to the iteration spaces of the kernels.

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 to compare the quality of monoparametrically tiled code with 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 algorithm to obtain a set of valid tiling hyperplanes. For five of these kernels, no legal tiling was found by Pluto, thus we ignore these kernels. We manually provide these tiling hyperplanes to our polyhedral compiler in order to replicate the tiling decision. The tile sizes used by the fixed-size code generator is 16.

In order to preserve the memory mapping, we apply the monoparametric tiling transformation to only 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]`. It is possible to also tile both the iteration space and the array accesses: we would obtain an internal polyhedral representation similar to the one we had in Section 4.1. However, because we would still need to use a quadratic function in order to fit with the original memory mapping, it would only degrade the performance, due to the additional branch conditions introduced by the tiled array accesses.

We did not expose parallelism in both generated tiled code, and we relied on the back-end compiler to do vectorization. Because we evaluate the performance of the monoparametric tiling transformation against the fixed-size tiling, we need two identical compilation flow, whose only difference is the nature of the tiling transformation used. Having a compilation flow which produces optimal performance (for example, by exposing parallelism) is not required to prove our claim.

The generated `C` code was compiled using gcc (version 6.3), with option `-O3` enabled. The problem sizes considered are the ones corresponding to the large dataset of Polybench, 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 used the nearest power of 2 as problem sizes and generated a simplified monoparametric tiled code in which we assume that the tile size parameter divides the problem size parameters.

The execution times[7] are shown in Figure 9. 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.

5 Related work

We already presented some of the fundamental work on tiling [24, 54] 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

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

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 final, 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. [43, 44] 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, 44]. Note 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 was then extended to manage imperfectly nested loops [26] one of the first parametrically tiled code generators for scanning unions of polyhedra.

Independently, Hartono et al. [22] presented a code generation scheme called *PrimeTile* which also manages imperfectly nested loops. 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. Because tile origins of different stripes are not aligned, we cannot find wavefront parallelism and this scheme cannot be adapted to generate parallel tiled code.

Later, Hartono et al. [23] presented a code generation scheme called *DynTile* to generate parallel tiled code for imperfectly nested loops. 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] 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 [47] 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. [44] (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. Grsslinger 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 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: partitioning, which is a reindexing introducing new tiling indices and constructing an equivalent program representation, and the tiling proper which changes the schedule of the program to satisfy the atomicity property of a tile. We showed that monoparametric partitioning transformation is a polyhedral transformation: it transforms a polyhedron into a union of $\mathbb{Z}$-polyhedra 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 polyhedral analyses and transformations after tiling, whereas in other approaches to parametric tiling, 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 most likely, 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 interfaced the C++ standalone library with a source-to-source C compiler and compared the quality of the fixed-sized tiled and the monoparametric tiled code generated, under a common compiler code generator.

References

[1] Acharya A, Bondhugula U (2015) Pluto+: Near-complete modeling of affine transformations for parallelism and locality. SIGPLAN Notices 50(8):54–64, DOI 10.1145/2858788.2688512

[2] Achtziger W, Zimmermann KH (2000) Finding quadratic schedules for affine recurrence equations via nonsmooth optimization. Journal of VLSI Signal Processing Systems 25(3):235–260, DOI 10.1023/A:1008139706909

[3] Alias C, Plesco A (2015) Data-aware Process Networks. Research Report RR-8735, Inria - Research Centre Grenoble – Rhône-Alpes, URL https://hal.inria.fr/hal-01158726

[4] Alias C, Baray F, Darte A (2007) 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)

[5] Amarasinghe SP (1997) Parallelizing compiler techniques based on linear inequalities. PhD thesis, Stanford University

[6] Bandishti V, Pananilath I, Bondhugula U (2012) Tiling stencil computations to maximize parallelism. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, SC '12, pp 1–11

[7] Baskaran MM, Hartono A, Tavarageri S, Henretty T, Ramanujam J, Sadayappan P (2010) Parameterized tiling revisited. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, New York, NY, USA, CGO '10, pp 200–209, DOI 10.1145/1772954.1772983

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

[9] Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society, Washington, DC, USA, PACT '04, pp 7–16, DOI 10.1109/PACT.2004.11

[10] Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '08, pp 101–113, DOI 10.1145/1375581.1375595

[11] Bu J, Deprettere EF, Dewilde P (1990) A design methodology for fixed-size systolic arrays. In: Application Specific Array Processors, 1990. Proceedings of the International Conference on, IEEE, pp 591–602

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

[13] Darte A, Schreiber R, Villard G (2003) Lattice-based memory allocation. In: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, ACM, New York, NY, USA, CASES '03, pp 298–308, DOI 10.1145/951710.951749

[14] Feautrier P (1991) Dataflow analysis of array and scalar references. International Journal of Parallel Programming 20(1):23–53, DOI 10.1007/BF01407931

[15] Feautrier P (1992) Some efficient solutions to the affine scheduling problem: I. one-dimensional time. International Journal of Parallel Programming 21(5):313–348, DOI 10.1007/BF01407835

[16] Feautrier P (1992) Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. International Journal of Parallel Programming 21(6):389–420, DOI 10.1007/BF01379404

[17] Feautrier P (2015) The power of polynomials. In: Jimborean A, Darte A (eds) 5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15), Amsterdam, Netherlands, pp 1–5

[18] Frigo M, Johnson SG (1998) Fftw: An adaptive software architecture for the fft. In: Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on, IEEE, vol 3, pp 1381–1384

[19] Grosser T, Zheng H, Aloor R, Simbürger A, Größlinger A, Pouchet LN (2011) Polly – polyhedral optimization in LLVM. In: Alias C, Bastoul C (eds) 1st International Workshop on Polyhedral Compilation Techniques (IMPACT), Chamonix, France, pp 1–6

[20] Grosser T, Cohen A, Holewinski J, Sadayappan P, Verdoolaege S (2014) Hybrid hexagonal/classical tiling for GPUs. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM, New York, NY, USA, CGO '14, pp 66–75, DOI 10.1145/2544137.2544160

[21] Grosslinger A, Griebl M, Lengauer C (2004) Introducing non-linear parameters to the polyhedron model. Tech. rep., Technische Universitt Mnchen

[22] Hartono A, Baskaran MM, Bastoul C, Cohen A, Krishnamoorthy S, Norris B, Ramanujam J, Sadayappan P (2009) Parametric multi-level tiling of imperfectly nested loops. In: Proceedings of the 23rd International Conference on Supercomputing, ACM, New York, NY, USA, ICS '09, pp 147–157, DOI 10.1145/1542275.1542301

[23] Hartono A, Baskaran M, Ramanujam J, Sadayappan P (2010) Dyntile: Parametric tiled loop generation for parallel execution on multicore processors. In: International Symposium on Parallel Distributed Processing (IPDPS),, pp 1–12, DOI 10.1109/IPDPS.2010.5470459

[24] Irigoin F, Triolet R (1988) Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'88, pp 319–329, DOI 10.1145/73560.73588

[25] Kim D, Rajopadhye S (2010) Efficient tiled loop generation: D-tiling. In: Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing, Springer-Verlag, Berlin, Heidelberg, LCPC'09, pp 293–307, DOI 10.1007/978-3-642-13374-9_20

[26] Kim D, Rajopadhye SV (2009) Parameterized tiling for imperfectly nested loops. Tech. Rep. CS-09-101, Colorado State University

[27] Kim D, Renganarayanan L, Rostron D, Rajopadhye SV, Strout MM (2007) 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, p 51, DOI 10.1145/1362622.1362691

[28] Kong M, Pop A, Pouchet LN, Govindarajan R, Cohen A, Sadayappan P (2015) Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. ACM Transactions on Architecture and Code Optimization 11(4):61:1–61:30, DOI 10.1145/2687652

[29] Krishnamoorthy S, Baskaran M, Bondhugula U, Ramanujam J, Rountev A, Sadayappan P (2007) Effective automatic parallelization of stencil computations. SIGPLAN conference of Programing Language Design and Implementation 42(6):235–244

[30] Lam MD, Rothberg EE, Wolf ME (1991) The cache performance and optimizations of blocked algorithms. In: ACM SIGARCH Computer Architecture News, ACM, vol 19, pp 63–74

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

[32] Loechner V (1999) Polylib: A library for manipulating parameterized polyhedra. URL https://repo.or.cz/polylib.git/blob_plain/HEAD:/doc/parampoly-doc.ps.gz

[33] Mauras C (1989) 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

[34] Nookala SPK, Risset T (2000) A library for Z-polyhedral operations

[35] Pop S, Cohen A, Bastoul C, Girbal S, Silber GA, Vasilache N (2006) GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In: Proceedings of the 4th GCC Developper's Summit, Ottawa, Ontario, Unknown or Invalid Region, pp 1–18

[36] Püschel M, Moura JM, Singer B, Xiong J, Johnson J, Padua D, Veloso M, Johnson RW (2004) Spiral: A generator for platform-adapted libraries of signal processing algorithms. The International Journal of High Performance Computing Applications 18(1):21–45

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

[38] Quilleré F, Rajopadhye S, Wilde D (2000) Generation of efficient nested loops from polyhedra. International Journal of Parallel Programming 28(5):469–498, DOI 10.1023/A:1007554627716

[39] Quinton P, Van Dongen V (1989) 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

[40] Quinton P, Rajopadhye SV, Risset T (1997) On manipulating Z-polyhedra using a canonical representation. Parallel Processing Letters 7:181–194

[41] Rajopadhye SV, Purushothaman S, Fujimoto RM (1986) On synthesizing systolic arrays from recurrence equations with linear dependencies. In: International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, pp 488–503

[42] Reed DA, Adams LM, Partick ML (1987) Stencils and problem partitionings: Their influence on the performance of multiple processor systems. IEEE Transactions on Computers 36(7):845–858

[43] Renganarayanan L, Kim D, Rajopadhye SV, Strout MM (2007) Parameterized tiled loops for free. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp 405–414, DOI 10.1145/1250734.1250780

[44] Renganarayanan L, Kim D, Rajopadhye SV, Strout MM (2012) Parameterized loop tiling. ACM Trans Program Lang Syst 34(1):3, DOI 10.1145/2160910.2160912

[45] Schreiber R, Dongarra JJ (1990) Automatic blocking of nested loops. Tech. rep., University of Tennessee

[46] Schrijver A (1986) Theory of Linear and Integer Programming. John Wiley & Sons, Inc., New York, NY, USA

[47] Tavarageri S, Hartono A, Baskaran M, Pouchet LN, Ramanujam J, Sadayappan P (2010) Parametric tiling of affine loop nests. In: 15th Workshop on Compilers for Parallel Computing (CPC'10), Vienna, Austria, pp 1–15

[48] Teich J, Thiele L (1993) Partitioning of processor arrays: A piecewise regular approach. Integration, the VLSI journal 14(3):297–332

[49] Trifunovic K, Cohen A, Edelsohn D, Li F, Grosser T, Jagasia H, Ladelsky R, Pop S, Sjödin J, Upadrasta R (2010) GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In: GCC Research Opportunities Workshop (GROW'10), Pisa, Italy

[50] Verdoolaege S (2010) isl: An integer set library for the polyhedral model. In: Fukuda K, Hoeven J, Joswig M, Takayama N (eds) Mathematical Software (ICMS'10), Springer-Verlag, LNCS 6327, pp 299–302

[51] Whaley RC, Dongarra JJ (1998) Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing, IEEE Computer Society, pp 1–27

[52] Wolf ME, Lam M (1991) A data locality optimizing algorithm. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Totonto, CA

[53] Wolfe M (1989) Iteration space tiling for memory hierarchies. In: Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp 357–361, URL http://dl.acm.org/citation.cfm?id=645818.669220

[54] Xue J (2000) Loop Tiling for Parallelism. Kluwer Academic Publishers, Norwell, MA, USA

[55] Yuki T, Gupta G, Kim D, Pathan T, Rajopadhye SV (2012) Alphaz: A system for design space exploration in the polyhedral model. In: Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, pp 17–31, DOI 10.1007/978-3-642-37658-0_2