On the Verification of Polyhedral Program Transformations

Christophe Alias
Inria, CNRS, ENS-Lyon, UCBL, Univ. Lyon, France
Christophe.Alias@inria.fr

Guillaume Iooss
Inria, France
Guillaume.Iooss@inria.fr

Sanjay Rajopadhye
Colorado State University, USA
Sanjay.Rajopadhye@colostate.edu

***Abstract*—This paper presents a pragma language to specify a polyhedral program transformation directly in the code and a verification algorithm able to check the correctness of the specified transformation. Our language is general enough to specify a loop tiling by an arbitrary polyhedral tile shape (e.g., hexagons, diamonds, trapezoids), and whose size may depend on a scaling parameter (monoparametric tiling). Our verification algorithm checks the legality of the proposed transformation, and provides counterexamples of unsatisfied dependences when it is incorrect. In addition, out tool infers the domain of scaling parameters where the tiling is not legal. We developed a tool suite implementing these concepts with a verification tool (MPPCHECK) and a code generation tool (MPPCODEGEN), that are available and may be downloaded together with a rich set of examples. We evaluate the performance of the verification and the code generation on kernels from the PolyBench suite.**

***Index Terms*—Polyhedral model, tiling, scheduling, verification**

I. Introduction

Loop tiling [19], [25] is an important program transformation which introduces granularity control in a loop nest. It groups iterations of a loop into sets that to be executed *atomically* (be that sequentially or in parallel) called *tiles*. Among its many uses are expressing coarse-grain parallelism across tiles, improving the locality or the operational intensity of a program. Tiling can also be applied several times on the same program, to create a hierarchy of tiles, where each level usually corresponds to a level in the memory/network hierarchy of the target architecture. For example, the matrix multiplication implementation in BLIS [33] uses up to three levels of tiling.

The tiling transformation has many variants, depending on the nature of the tile sizes (constant, monoparametric [18] or parametric) and the tile shape. The most commonly used tile shape is the parallelogram (more precisely, hyper-parallelepipeds), defined by their hyperplanes. Other tile shapes include trapezoids [22] or hexagons [14], [15].

Due to the variation in the code structure, the tiling transformation usually requires a separate implementation per tile shape. The rectangular case is the easiest to implement (using strip-mining and loop interchange), but some of them are not so simple. To our knowledge, no compiler supports simultaneously all of these tiles shapes, and the option of the different tile sizes (fixed, or (mono) parametric). This prevents the comparison of the efficiency of tiling using different tile shapes, by restricting the available optimization space.

The basic conditions on the legality of tiling are well understood. The fact that tiles are atomic implies that there cannot be any circular dependences between tiles. When a user proposes a tiling or other program transformation, an automatic legality checker is crucial, because it is possible that the proposed transformation is not legal and either violates data dependences or introduces cycles among tiles.

In this paper, we present a way to specify a program transformation directly in the code, through pragmas attached to program statements. In particular, we may specify any fixed-size and monoparametric tiling using any polyhedral shape (e.g., hexagons, diamonds, trapezoids). We also allow pragmas to specify affine schedules and parallel dimensions. We provide a verification algorithm that checks the validity of the specified transformation. It also generates counterexamples to help the debugging process when the proposed transformation is incorrect, and the domain of invalid tile *scaling* parameter values, when monoparametric tiling is specified. Our specific contributions are as follows.

- We propose a pragma language able to specify any affine schedule in a program. In particular, our language is expressive enough to describe loop tilings using any polyhedral tile shape. The tile size can either be constant, or can depend on a scaling parameter [18].
- A verification algorithm to check the correctness of the specified schedule. In case of failure, our algorithm lists several instances of unsatisfied dependence instances to help the debugging. Also, the set of incorrect values for the scaling parameter is inferred, from which a validity domain may be deduced by polyhedral subtraction.
- A complete tool suite for schedule verification (MPPCHECK[1]) and code generation (MPPCODEGEN[2]) from our specification pragma language.

The remainder of this paper is structured as follows. Section II introduces the polyhedral model. Section III describes how polyhedral loop transformations are specified in our language. Section IV presents our verification algorithm. Section V gives the experimental results obtained on the kernels from Polybench/C. Section VI discusses the related work. Finally, Section VII concludes this paper and draws future research directions.

[1]http://foobar.ens-lyon.fr/mppcheck
[2]http://foobar.ens-lyon.fr/mppcodegen

Author version - Published in 18th International Conference on High Performance Computing & Simulation
3rd Special Session on Compiler Architecture, Design and Optimization, 2020

II. Preliminaries

This section outlines the concepts of polyhedral compilation used in this paper. In particular, we recall monoparametric tiling with general tile shapes, the main transformation addressed in this paper.

A. Polyhedral model

The polyhedral model [8]–[11], [29], [31] is a general framework to design loop transformations, historically geared towards source-level automatic parallelization [11] and data locality improvement [5]. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [8], scheduling [10] or loop tiling [5] to quote a few) . The polyhedral model manipulates program fragments consisting of nested `for` loops and conditionals manipulating arrays and scalar variables, such that loop bounds, conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters (input sizes, e.g., N)). Thus, the control is static and may be analysed at compile-time. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i} \rangle$. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. Figure 2.(a) provides the iteration domains $\mathcal{D}_S = \{i \mid 0 \leq i < N\}$ and $\mathcal{D}_T = \{(i,j) \mid 0 \leq i,j < N\}$ for the matrix-vector kernel presented later.

B. Dependences

Given an operation (ie., an instance of some statement in the program) ω, we write $\mathrm{read}(\omega)$ (resp. $\mathrm{write}(\omega)$) the set of addresses read (resp. written) by ω. There exists a *dependence* from an operation s to an operation t iff $s \prec_{seq} t$, both operations access the same address and one access is a write. When $\mathrm{write}(s) \cap \mathrm{read}(t) \neq \emptyset$ (resp. $\mathrm{read}(s) \cap \mathrm{write}(t) \neq \emptyset$, $\mathrm{write}(s) \cap \mathrm{write}(t) \neq \emptyset$), we have a *flow* dependence and we write: $s \longrightarrow^{\text{FLOW}} t$ (resp. *anti*: $s \longrightarrow^{\text{ANTI}} t$, *output*: $s \longrightarrow^{\text{OUTPUT}} t$). Dependences are usually represented by a *reduced dependence graph* $\mathcal{G} = (\mathcal{S}, \Delta)$, whose nodes are the program statements; and edges $S \xrightarrow{\Delta_{ST}}^{\ell} T$ are labelled by $\Delta_{ST} = \{(\vec{i}, \vec{j}) \mid \langle S, \vec{i} \rangle \rightarrow^{\ell} \langle T, \vec{j} \rangle\}$, where $\ell \in \{\text{FLOW}, \text{ANTI}, \text{OUTPUT}\}$. For convenience, the union of flow, anti and output dependences is denoted by $\rightarrow$. The dependence relation of a program is *transitively closed*: if $\langle \rightarrow \rangle$ denotes the transitive closure of $\rightarrow$, then $\langle \rightarrow \rangle = \rightarrow$.

C. Scheduling & Tiling

A *schedule* θ_S assigns each operation $\langle S, \vec{i} \rangle$ with a timestamp $\theta_S(\vec{i}) \in (\mathbb{Z}^d, \ll)$. Intuitively, $\theta_S(\vec{i})$ is the iteration of $\langle S, \vec{i} \rangle$ in the transformed program. A schedule is *correct* if $\langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j})$, the lexicographic order ensuring that the dependence is preserved. We also have schedules where some dimensions are marked *explicitly parallel*. We extend $\ll$ to be strict equality in these dimensions. This ensures that two iterations with identical outer timestamps are executed by the same (virtual) processor.

Tiling is a reindexing transformation which groups iteration into tiles to be executed atomically. There are many variants of this transformation. In this paper, we consider *affine tiling with constant tile size* and *monoparametric tiling*.

Rectangular tiling reindexes any iteration $\vec{i} \in \mathcal{D}_S$ to an iteration $(\vec{i}_{block}, \vec{i}_{local})$ such that $\vec{i} = \mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local})$, with $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = (\mathrm{diag}\,\vec{s})\ \vec{i}_{block} + \vec{i}_{local}$, $0 \leq \vec{i}_{local} < \vec{s}$ where $\vec{s}$ is a vector collecting the tile size across each dimension of the iteration domain. $\vec{i}_{block}$ is called the *outer* tile iterator and $\vec{i}_{local}$ is called the *inner* tile iterator. The companion schedule associated to the tiling $\theta_S(\vec{i}_{block}, \vec{i}_{local})$ orders $\vec{i}_{block}$ first to to ensure the execution tile by tile. Figure 2.(b) gives an example of rectangular tiling with $\vec{s} = (2,2)$. To enforce the atomicity (avoid cross dependences between two tiles), it is sometimes desirable to precede the tiling by an injective affine mapping ϕ_S. The coordinates of $\phi_S(\vec{i})$, for $\vec{i} \in \mathcal{D}_S$ are usually called *tiling hyperplanes*. In that case, the transformation $\mathcal{T}_S^{-1} \circ \phi_S$ for some statements S is called an *affine tiling*. Note that rectangular tiling is a particular case of affine tiling where ϕ_S is the identity mapping.

When $\vec{s}$ depends on a scaling parameter $b \geq 1$, $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = b.(\mathrm{diag}\,\vec{r}).\vec{i}_{block} + \vec{i}_{local}$, where $\vec{r}$ is a constant vector called the *ratio*, the tiling is said *monoparametric parallelepipedic*. Finally, when the tile shape is an arbitrary convex polyhedron $b.\mathcal{P}$, whose size depends on a scaling parameter $b \geq 1$, the tiling is said *monoparametric general*. In that case, $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = b.L\vec{i}_{block} + \vec{i}_{local}$, where $\vec{i}_{local} \in b.\mathcal{P}$ and the matrix $b.L$ defines a linear lattice spanning the tile origins. In both cases, monoparametric tiling is a *polyhedral transformation* [18]: the transformed domain and index functions can still be expressed in Presburger arithmetic.

III. Specifying a polyhedral transformation

This section outlines our pragma language to specify a polyhedral transformation directly in the code. First, we explain how to specify a simple affine schedule. Then, we show how to express loop tiling. Through examples, we describe the output of our verification tool, MPPCHECK.

A. Affine scheduling

The program is enclosed with pragmas `begin_scop` and `end_scop`. Then, we specify an affine schedule per statement with the pragmas `schedule`, using an array-style syntax. On the matrix-vector example, the parallel schedule $\theta_S(i) = 0$ and $\theta_T(i,j) = j + 1$ would be specified as:

```
#pragma begin_scop
  for(i=0; i<N; i++)
    {
#pragma schedule[0]
      y[i] = 0; //S
      for(j=0; j<N; j++)
#pragma schedule[j+1]
        y[i] = y[i] + a[i][j]*x[j]; //T
    }
#pragma end_scop
```

On that example, MPPCHECK would simply say `PASSED`, meaning that the schedule is correct. However, if we specify $\theta_T(i,j) = j$, the schedule is no longer correct, as the parallel initializations $\langle S, i\rangle$ would overlap with the first parallel iterations $\langle T, i, 0\rangle$, for $0 \leq i < N$.
In that case, MPPCHECK emits an error message with an example of unsatisfied dependence:

```
ERROR: dependence not satisfied:
#0 --[FLOW]--> #1, read #1, depth 1
y[i]=(0) ---> y[i]=y[i]+(a[i][j]*x[j])
When
/
| -1+N >= 0
\

#0 i=0  --[FLOW]--> #1 i=0 j=0
```

B. Affine tiling

Tiling hyperplanes are specified with the pragma `tile_hyperplanes` in the same way as schedules. They are completed by the tile size across each hyperplane in their specification order (pragma `tile_size`). Finally the schedule is enhanced with outer tile iterators, denoted by `__T`, again for each hyperplane in their specification order:

```
#pragma begin_scop
  for(i=0; i<N; i++)
    {
#pragma tile_hyperplanes[i][0]
#pragma tile_size[8][4]
#pragma schedule[__T][__T][0][i]
      y[i] = 0; //S

      for(j=0; j<N; j++)
#pragma tile_hyperplanes[i][j]
#pragma tile_size[8][4]
#pragma schedule[__T][__T][1][i][j]
        y[i] = y[i] + a[i][j]*x[j]; //T
    }
#pragma end_scop
```

Any complex loop tiling structure (e.g. mixing tiled/non-tiled loops) may be expressed by preceding/interleaving `__T` dimensions with non `__T` dimensions.

Again, if the schedule is incorrect, for instance if we specify the tiling hyperplanes $(i, -j)$ for the statement T, MPPCHECK emits an error message and lists each unsatisfied dependence together with a bad instance:

```
ERROR: dependence not satisfied:
#1 --[FLOW]--> #1, read #1, depth 1
y[i]=y[i]+(a[i][j]*x[j]) ---> y[i]=y[i]+(a[i][j]*x[j])
When
/
| -2+N >= 0
| -1+N >= 0
\

#1 i=0 j=0 tile_counter_0=0 tile_counter_1=0
--[FLOW]-->
#1 i=0 j=1 tile_counter_0=0 tile_counter_1=-1
[...]
```

Here, we clearly see that the dependence goes backward across the second tiling hyperplane $(-j)$, as `tile_counter_1` decreases (from 0 to -1). Hence we may conclude that the hyperplane $-j$ is faulty.

C. Monoparametric parallelepipedic tiling

The monoparametric parallelepipedic tiling (affine tiling with tile sizes depending on a scaling parameter) is specified in the same way as affine tiling with constant tile size. The only difference is that the pragma `tile_size` is replaced by a pragma `tile_ratio` specifying the tile ratio across each tiling hyperplane:

```
#pragma begin_scop
  for(i=0; i<N; i++)
    {
#pragma tile_hyperplanes[i][0]
#pragma tile_ratio[2][1]
#pragma schedule[__T][__T][0][i]
      y[i] = 0;

      for(j=0; j<N; j++)
#pragma tile_hyperplanes[i][j]
#pragma tile_ratio[2][1]
#pragma schedule[__T][__T][1][i][j]
        y[i] = y[i] + a[i][j]*x[j];
    }
#pragma end_scop
```

When the schedule is incorrect, MPPCHECK is able to infer – in addition to unsatisfied dependence instances – the domain of incorrect values for the tile size scaling parameter (denoted by `block_size`).
Again, with the $-j$ hyperplane, we get:

```
Tiling is incorrect when:
/
| -1+block_size >= 0
\
```

We can deduce a domain of correct scaling parameter from this. Here, the tiling is definitely wrong: no `block_size` value can lead to a correct tiling.

D. Monoparametric general tiling

An important feature is the ability to specify monoparametric tilings with general convex tile shape. Here is an example of hexagonal tiling on the jacobi-1D (perfect) kernel, as depicted on Figure 1:

```
#pragma begin_scop
  for (t = 1; t <= TSTEPS; t++)
    for (i = 1; i < N - 1; i++)
#pragma tile_lattice[1][0][-3][6]
#pragma tile_shape_closed[i-t][t+1][t+i]
#pragma tile_shape_open[-i+t+4][1-t][4-t-i]
#pragma schedule[__T][__T][t][i]
      A[t][i] = 0.3 * (A[t-1][i-1]
                       + A[t-1][i]
                       + A[t-1][i+1]);
#pragma end_scop
```

The tiling features are specified per statement, in the same way as for parallelepipedic tiling. For each statement, the following elements must be specified:

- The lattice L of tile origins (pragma `tile_lattice`) given *line by line*, here $L = \begin{pmatrix} 1 & 0 \\ -3 & 6 \end{pmatrix}$. An additional line may specify the divisors per column

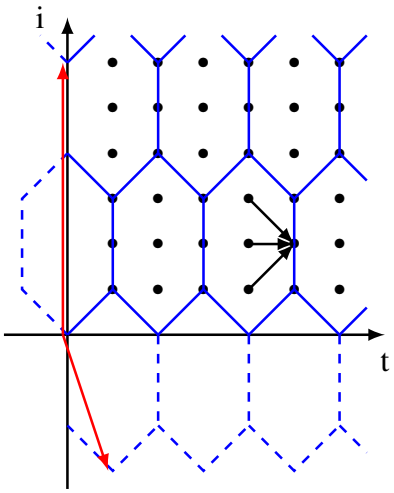


Fig. 1. Hexagonal tiling for the Jacobi 1D kernel. In blue are the hexagonal tiles, dotted for the parts outside of the iteration space. The red arrows are the two vectors of the lattice of tile origins. The black arrow in the center of the figure are the dependences of the Jacobi1D kernel. In this example, we took $b = 1$.

of L. For instance, appending `[2][1]` would specify $L = \begin{pmatrix} 1/2 & 0 \\ -3/2 & 6 \end{pmatrix}$.

- The tile shape $\mathcal{P}$, as a conjunction of *closed constraints* (affine form ≥ 0, `tile_shape_closed`), here $i - t \geq 0$, $t + 1 \geq 0$, $t + i \geq 0$; and *open constraints* (affine form > 0, `tile_shape_open`), here $-i + t + 4 > 0$, $1 - t > 0$, $4 - t - i > 0$.
- The schedule, in the same way as for affine tiling. Recall that the tile counters $\vec{i}_{block}$ (outer) and $\vec{i}_{local}$ (inner) for an original iteration $\vec{i}$ are such that: $\vec{i} = b.L\vec{i}_{block} + \vec{i}_{local}$, with $\vec{i}_{local} \in b.\mathcal{P}$ and b the scaling parameter (denoted as `block_size` above). Here the `__T` denotes the outer tile counters $\vec{i}_{block}$, as constrained by the lattice of origins. `t` and `i` are – as there name suggest – part of the original iteration vector.

With that lattice, the tile origins verify:

$$\begin{pmatrix} t_0 \\ i_0 \end{pmatrix} = b.\begin{pmatrix} 1 \\ -3 \end{pmatrix} i_{block_0} + b.\begin{pmatrix} 0 \\ 6 \end{pmatrix} i_{block_1}$$

Hence, the second outer tile counter (i_{block_1}, variable `tile_counter_1`) will iterates through a layer of hexagons along the i axis, while the first outer counter (i_{block_0}, variable `tile_counter_0`) iterates through the layers along the t axis. This makes possible to implement concurrent start parallelism, provided a correct data privatization.

E. ... and composition thereof

All these constructions may coexist in a specification. For instance, we can have together statements with parallelepipedic tiling (and possibly different tile size/ratio), statements with general tiling shapes, and non-tiled statements.

IV. Verifying a polyhedral transformation

This section describes our verification algorithm. We first show how non-tiled affine scheduling and constant-size tiling may be directly checked. Then we present a method to check a monoparametric tiling.

A. Non-tiled programs

When the program is not tiled, it suffices to check the satisfiability of:

$$\mathcal{C} := \forall \langle S, \vec{i}\rangle, \langle T, \vec{j}\rangle : \langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j}) \quad (1)$$

We can get rid of the universal quantifier ($\forall$) by checking the the negation:

$$\neg\mathcal{C} := \exists \langle S, \vec{i}\rangle, \langle T, \vec{j}\rangle : \langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \wedge \neg\left(\theta_S(\vec{i}) \ll \theta_T(\vec{j})\right)$$

Which is an existentially quantified composition of conjunctions and disjunctions (due to the lexicographic order $\ll$) of affine constraints whose satisfiability may be checked with state-of-the-art linear programming tools [7], [13]. When UNSAT $\neg\mathcal{C}$, the schedule is correct. When SAT $\neg\mathcal{C}$, however, the schedule is not correct and we may pick a counterexample in $\neg\mathcal{C}$.

B. Affine tiling

When the program is tiled, iteration domains are reindexed ($\vec{i} \mapsto (\vec{i}_{block}, \vec{i}_{local})$) and schedules are given on the reindexed domain ($(\vec{i}_{block}, \vec{i}_{local})$, sometimes $(\vec{i}_{block}, \vec{i})$). Equation (1) becomes:

$$\begin{aligned} \mathcal{C} := \quad & \forall \langle S, \vec{i}\rangle, \langle T, \vec{j}\rangle : \\ & \langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \wedge \\ & \mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = \vec{i} \wedge \mathcal{T}_T(\vec{j}_{block}, \vec{j}_{local}) = \vec{j} \\ & \wedge\ 0 \leq \vec{i}_{local}, \vec{j}_{local} < \vec{s} \\ & \Rightarrow \theta_S(\vec{i}_{block}, \vec{i}_{local}) \ll \theta_T(\vec{j}_{block}, \vec{j}_{local}) \end{aligned} \quad (2)$$

When the tile size $\vec{s}$ is constant, $\mathcal{T}$ is affine per statement, hence we may check $\neg\mathcal{C}$ in the same way as for non-tiled statements. Note that neither tiled iteration domains nor tiled dependence relations are involved in this formulation: the only connection with the tiling world is the tiling function $\mathcal{T}$.

C. Monoparametric tiling

As soon as a tile size depends on a parameter, $\mathcal{T}_S$ is no longer affine. On the monoparametric parallelepipedic case we get stuck with the quadratic expression $b.(\mathrm{diag}\,\vec{r}).\vec{i}_{block}$ in $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = b.(\mathrm{diag}\,\vec{r}).\vec{i}_{block} + \vec{i}_{local}$ (recall that $\vec{r}$ is a constant vector and $b \geq 1$ is a parameter). On the monoparametric general case we get stuck with the quadratic expression $b.L\vec{i}_{block}$ in $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = b.L\vec{i}_{block} + \vec{i}_{local}$ (recall that L is a constant non-singular matrix).

Hence, we need to express the dependence relation directly on the indexed domain. First, we show how to turn a reduced dependence graph to an equivalent reduced dependence graph where the dependence relations Δ_{XY} are *guarded affine functions*. We prove (Theorem 4.1) that the obtained reduced dependence graph is equivalent to the original. Then, we show how to tile these guarded functions.

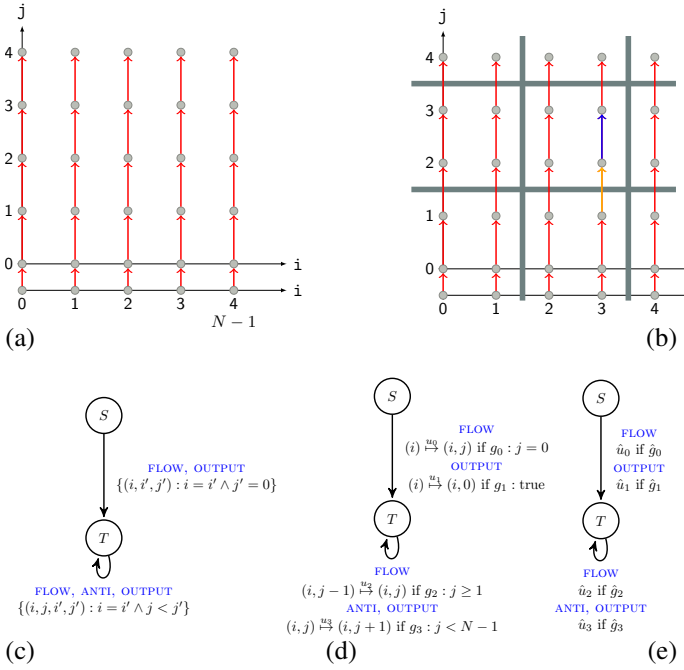


Fig. 2. Tiling the reduced dependence graph. The reduced dependence graph (c) is normalized with guarded dependences functions (d). In turn, guards and functions are tiled (e).

Step 1. Turn dependence relations to guarded functions: For each type of dependence, we build guarded affine functions in the following way. *Flow-dependences* are specified by a function source_k, mapping an operation $\langle T, \vec{j} \rangle$ to the last operation executed before $\langle T, \vec{j} \rangle$, which writes its k-th read address: $\text{source}_k(\langle T, \vec{j} \rangle) \longrightarrow^{\text{FLOW}} \langle T, \vec{j} \rangle$ for any $\vec{j} \in \mathcal{D}_T$. The source function is always computable, the result is a piecewise affine mapping [8] whose pieces are called *branches*. Figure 2.(c) depicts the dependence graph for the matrix-vector product. (d) is the same graph with the source function $\text{source}_1(\langle T, i, j \rangle)$ for the first read, $y[i]$, defined by branch $\langle S, i \rangle$ when $j = 0$, and branch $\langle T, i, j-1 \rangle$ when $j \geq 1$.

Similarly, *output-dependences* are specified by a function NextWrite, mapping an operation to the next operation writing the same address, in the original execution order: $\langle S, \vec{i} \rangle \longrightarrow^{\text{OUTPUT}} \text{NextWrite}(\langle S, \vec{i} \rangle)$ for any $i \in \mathcal{D}_S$. Finally, *anti-dependences* are specified by a function $\text{NextWriteForRead}_k$ mapping an operation to the next write to the address accessed by its kth read: $\langle S, \vec{i} \rangle \longrightarrow^{\text{ANTI}} \text{NextWriteForRead}_k(\langle S, \vec{i} \rangle)$ for any $i \in \mathcal{D}_S$ and any read number k. The functions NextWrite and NextWriteForRead may be computed with the same algorithm as that to compute the source function, with a few modifications to seek a next write instead of a previous read. Again, the result is a piecewise affine mapping with possibly multiple branches. Figure 2.(d) show the new reduced dependence graph with these functions, similar to the source function on that example. We end-up with 5 branches: three branches from T to T with type labels FLOW, ANTI and OUTPUT ; and two branches from S to T with type labels FLOW and OUTPUT. All theses dependences form a *subset* of the dependence relation. Still, they are sufficient to describe all the dependences by transitivity, as shown by the following theorem.

Theorem 4.1: Consider a program and its dependence relation δ. Let δ' be the smallest dependence relation such as:

- $\text{source}\langle S, \vec{i} \rangle \longrightarrow^{\text{FLOW}} \langle S, \vec{i} \rangle \in \delta'$, for any $\langle S, \vec{i} \rangle \in \text{dom source}$
- $\langle S, \vec{i} \rangle \longrightarrow^{\text{OUTPUT}} \text{NextWrite}\langle S, \vec{i} \rangle \in \delta'$ for any $\vec{i} \in \mathcal{D}_S$.
- $\langle S, \vec{i} \rangle \longrightarrow^{\text{ANTI}} \text{NextWriteForRead}_k\langle S, \vec{i} \rangle \in \delta'$ for any $\vec{i} \in \mathcal{D}_S$ and any read k.

Then, the transitive closure of δ' is δ, $\langle \delta' \rangle = \delta$

Proof.

- Since $\delta' \subseteq \delta$, we have $\langle \delta' \rangle \subseteq \langle \delta \rangle$. Also, $\langle \delta \rangle = \delta$, since the dependence relation of a program is transitively closed. Hence $\langle \delta' \rangle \subseteq \delta$.
- We now prove that $\delta \subseteq \langle \delta' \rangle$
  - Let $W \longrightarrow^{\text{OUTPUT}} W' \in \delta$. Since $W \prec_{seq} W'$, $W' = \text{NextWrite}^p(W)$ for some p. Hence $W \longrightarrow^{\text{OUTPUT}} W' \in \langle \delta' \rangle$.
  - Let $W \longrightarrow^{\text{FLOW}} R \in \delta$.
    If $W = \text{source}_k\, R$ for some read k, $W \longrightarrow^{\text{FLOW}} R \in \delta' \subseteq \langle \delta' \rangle$.
    If $W \neq \text{source}_k\, R$ for any read k, let W_0 be the source for a read of R whose address is written by W. $W \prec_{seq} W_0$, otherwise W_0 would not be a source for $\text{write}(W)$. Then: $W \longrightarrow^{\text{OUTPUT}} W_0 \in \delta$.
    Hence the chain:
    $W \overset{\langle \delta' \rangle}{\longrightarrow}{}^{\text{OUTPUT}} W_0 \overset{\delta'}{\longrightarrow}{}^{\text{FLOW}} R.$
    Thus $W \longrightarrow^{\text{FLOW}} R \in \langle \delta' \rangle$.
  - Let $R \longrightarrow^{\text{ANTI}} W \in \delta$.
    Let W_0 be the next write for the read of R whose address is written by W.
    If $W = W_0$, then $R \longrightarrow^{\text{ANTI}} W \in \delta' \subseteq \langle \delta' \rangle$.
    If $W \neq W_0$, then $W_0 \prec_{seq} W$, otherwise W_0 would not be the next write. Hence $W_0 \longrightarrow^{\text{OUTPUT}} W \in \delta$.
    Hence the chain:
    $R \overset{\delta'}{\longrightarrow}{}^{\text{ANTI}} W_0 \overset{\langle \delta' \rangle}{\longrightarrow}{}^{\text{OUTPUT}} W.$
    Thus: $R \longrightarrow^{\text{ANTI}} W \in \langle \delta' \rangle$ ■

Step 2. Tile the obtained reduced dependence graph: We transform each dependence branch to operate directly on the reindexed tiled domain. Consider a flow dependence branch $\langle S, u(\vec{j}) \rangle \longrightarrow^{\text{FLOW}} \langle T, \vec{j} \rangle$, when $\vec{j} \in \mathcal{D}$. If $\mathcal{T}_S(\vec{i}_{block}, \vec{i}_{local}) = \vec{i}$ and $\mathcal{T}_T(\vec{j}_{block}, \vec{j}_{local}) = \vec{j}$, then the version of u operating in the tiled domain is $\hat{u} = \mathcal{T}_S^{-1} \circ u \circ \mathcal{T}_T$ and the tiled guard is $\hat{\mathcal{D}} = \mathcal{T}_T^{-1}(\mathcal{D})$. Hence the tiled dependence branch $\langle S, \hat{u}(\vec{j}_{block}, \vec{j}_{local}) \rangle \longrightarrow^{\text{FLOW}} \langle T, \vec{j}_{block}, \vec{j}_{local} \rangle$, when $(\vec{j}_{block}, \vec{j}_{local}) \in \hat{\mathcal{D}}$. We tile ANTI and OUTPUT dependence in the same fashion.

We already proposed an algorithm, in the preprint [18], to tile a polyhedron $\mathcal{D} \mapsto \hat{\mathcal{D}}$ and an affine function $u \mapsto \hat{u}$. We also proved the polyhedral closure of parallelepipedic and general monoparametric tiling: $\hat{\mathcal{D}}$ is always a union

of polyhedra and $\hat{u}$ is always a piecewise affine mapping, both expressed in the Presburger arithmetic (affine without parametric coefficients). That way, any dependence graph is monoparametrically tilable.

Back to the matrix-vector example, $\hat{\mathcal{D}}_S$ is the set of iterations $(i_{block}, j_{block}, i_{local}, j_{local})$ satisfying:

$$[j_{block} = j_{local} = 0 \wedge i_{block} = N_{block} \wedge 0 \leq i_{local} < N_{local}] \bigvee$$
$$[j_{block} = j_{local} = 0 \wedge 0 \leq i_{block} < N_{block} \wedge 0 \leq i_{local} < b]$$

where $N = b.N_{block} + N_{local}, 0 \leq N_{local} < b$. The first conjunction set represents the full tiles (e.g. iterations (0,1) and (2,3) on Figure 2.(b)) while the second conjunction set represents the border tiles (e.g. iteration 4). $\hat{D}_T$ consists of 3 conjunction sets describing the corner cases (e.g. $i = 4$, $j = 4$, and both) and 1 conjunction set describing the full tiles. Finally, $\hat{u}$ is a piecewise affine mapping two branches depending if the source belong to the same tile (e.g. blue arrow) or not (e.g. yellow arrow).

Once the reduced dependence graph is tiled, we check the correctness of the schedule by solving UNSAT $\neg\mathcal{C}$. When the schedule is wrong, we project $\neg\mathcal{C}$ on b to get the domain of bad values for the scaling parameter, then we deduce the good ones by a simple subtraction from $\{b \mid b \geq 1\}$.

V. Experimental Evaluation

This section presents the experimental results obtained on the benchmarks of the polyhedral community.

A. Experimental Setup

We have implemented a tool, MPPCHECK[3], with our verification algorithm. Also, we have implemented a tiling code generator in a separate tool, MPPCODEGEN[4]. MPPCODEGEN computes the tiling from the specification and uses `iscc`'s code generator [35] to produce the final tiled code. We have applied our verification algorithm and our code generator on the kernels of PolyBench/C v3.2 [28], a benchmark suite with compute-intensive linear algebra kernels from the polyhedral compilation community. The experiments were run on a laptop with an Intel core i5 540M processor with 3GB DDR, except for the kernels `heat-3d` and `h3d-perf Hex` kernel (requiring more memory), where an AMD Opteron(TM) Processor 6272 32GB RAM was used.

Table 3 depicts the results. By default, a monoparametric parallelepipedic tiling was applied, except for the kernels suffixed with `Hex` (resp. `Diam`) where an *hexagonal* (resp. *diamond*) monoparametric general tiling was applied:

- **j1d-perf Diam** is a perfect loop nest variant of the `jacobi-1D` kernel with a diamond tiling:
  $L = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$
  $\mathcal{P} = \{(t,i) \mid t+i \geq 0, t-i \geq 0, 2-t-i > 0, 2+i-t > 0\}$.
- **j1d-perf Hex** is the same `jacobi-1D` variant with an hexagonal tiling:
  $L = \begin{pmatrix} 1 & 0 \\ -3 & 6 \end{pmatrix}$
  $\mathcal{P} = \{(t,i) \mid i-t \geq 0, t+1 \geq 0, t+i \geq 0, -i+t+4 > 0, 1-t > 0, 4-t-i > 0\}$.
- **j2d-perf Hex** is a perfect loop nest variant of the `jacobi-2D` kernel with an hybrid hexagonal/parallelepipedic tiling (tube with an hexagonal section along (t,i), directed towards j and sliced with an hyperplane $t+j$):
  $L = \begin{pmatrix} 1 & 0 & 0 \\ -3 & 6 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
  $\mathcal{P} = \{(t,i,j) \mid i-t \geq 0, t+1 \geq 0, t+i \geq 0, t+j \geq 0, -i+t+4 > 0, 1-t > 0, 4-t-i > 0, 1-t-j > 0\}$.
- **h3d-perf Hex** is a perfect loop nest variant of the `heat-3D` kernel with an hybrid hexagonal/parallelepipedic tiling (4D tube with an hexagonal section along (t,i), directed towards j,k and sliced with hyperplanes $t+j$ and $t+k$):
  $L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -3 & 6 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
  $\mathcal{P} = \{(t,i,j,k) \mid i-t \geq 0, t+1 \geq 0, t+i \geq 0, t+j \geq 0, t+k \geq 0, -i+t+4 > 0, 1-t > 0, 4-t-i > 0, 1-t-j > 0, 1-t-k > 0\}$.

For each kernel, we provide the number of iteration domains in the original program (column *Domains (based)*), the cumulated number of polyhedra after tiling each iteration domain, $\sum_S \operatorname{card} \hat{\mathcal{D}}_S$ (column *Domains (tiled)*), and the time spent to tile all the iteration domains (column *Build time*). Then, we provide the number of branches and the build time for the reduced dependence graph of the original program (columns *DG base*) and after tiling (columns *DG (tiled)*). Note that the number of branches for a tiled dependence is the number of pieces of the tiled dependence function $\hat{u}$. Moreover, we give the time spent by our verification algorithm itself (checking SAT $\mathcal{C}$), and the total time. Finally, we get the total time spent by MPPCODEGEN to generate the code, this includes iteration domain tiling (column *CodeGen*). By default, the timings are given in seconds.

B. Results

In most cases, our verification algorithm succeeds to passed the kernels in a reasonable amount of time. Not surprisingly, hexagonal tiling boils down to complex iteration domains and dependence functions with many corner cases. This impacts directly the checking time, as the computation of $\mathcal{C}$ involves, for each tiled dependence $S \to T$, the enumeration of all the tuples (source polyhedron $\in \hat{\mathcal{D}}_S$, dependence branch of $\hat{u}$, target polyhedron $\in \hat{\mathcal{D}}_T$). In particular, this explains the time spent on the `h3d-perf Hex` kernel.

The same remarks apply to our code generator, MPPCODEGEN. We were able to generate the code for all the considered kernels, the time spent is less than for verification, as code generation does not require the tiling of dependence functions.

[3] MPPCHECK is available at http://foobar.ens-lyon.fr/mppcheck
[4] MPPCODEGEN is available at http://foobar.ens-lyon.fr/mppcodegen

Kernel	Domain tiling			DG (base)		DG (tiled)		Verif		CodeGen
	Build time	Domains (base)	Domains (tiled)	Build time	Branches	Build time	Branches	Checking time	Total time	Total time
gemm	0.09	2	12	0.04	6	0.04	12	0.11	0.28	1.64
gemver	0.04	4	14	0.05	12	0.04	19	0.05	0.18	0.48
gesummv	0.04	5	14	0.04	14	0.04	22	0.05	0.17	0.38
symm	0.12	4	24	0.09	13	0.16	22	0.25	0.62	1.40
syr2k	0.06	2	12	0.04	6	0.04	9	0.08	0.22	0.79
syrk	0.06	2	12	0.04	6	0.04	9	0.08	0.22	0.74
trmm	0.06	2	12	0.05	8	0.05	13	0.14	0.30	0.76
2mm	0.23	4	36	0.09	13	0.10	27	0.46	0.88	1'3
3mm	0.68	6	78	0.14	19	0.17	58	2.71	3.70	1'24
atax	0.05	4	18	0.03	11	0.05	20	0.11	0.24	1.36
bicg	0.03	4	12	0.03	10	0.04	16	0.06	0.16	1.49
doitgen	0.43	3	32	0.10	7	0.04	11	0.18	0.74	4.29
mvt	0.02	2	8	0.02	6	0.03	12	0.03	0.09	0.24
cholesky	0.10	4	18	0.19	23	0.95	31	0.57	1.81	0.61
gramschmidt	0.18	7	32	0.15	24	0.22	39	0.27	0.82	2.08
lu	0.07	3	20	0.20	18	0.81	28	0.56	1.64	0.78
trisolv	0.02	3	8	0.03	14	0.11	18	0.10	0.26	0.27
correlation	0.34	15	48	0.32	45	0.33	57	0.34	1.33	3.66
covariance	0.18	8	32	0.23	35	0.38	43	0.35	1.15	1.47
floyd-warshall	0.04	1	8	0.07	23	0.23	38	0.54	0.87	1.60
fdtd-2d	1.51	4	160	0.18	27	2.27	132	9.63	13.59	11.18
heat-3d	9'23	2	940	0.81	42	4.57	199	3'1	12'29	22'22
h3d-perf Hex	45'2	1	1154	0.56	10	39.7	16038	4h44'6	5h29'49	59'44
jacobi-1d	0.15	2	42	0.04	14	0.14	38	0.82	1.15	1.69
j1d-perf Diam	0.83	1	37	0.02	3	0.10	48	1.72	2.67	25.39
j1d-perf Hex	1.28	1	54	0.02	3	0.21	60	2.41	3.92	44.34
jacobi-2d	3.75	2	134	0.14	22	0.50	112	8.94	13.33	16.59
j2d-perf Hex	41.70	1	273	0.07	5	1.24	918	2'2	2'45	4'10
seidel-2d	1.31	1	74	0.14	19	1.47	84	13.67	16.59	4.56

Fig. 3. Compilation time (in seconds) of the tiling transformation, the dependence graph construction (both on the non-tiled program, and the tiled program) and tiling verification analysis. We also provide the number of branches of both dependence graphs. By default, a monoparametric parallelepipedic tiling is used. "Hex" means that a 45° hexagonal tiling was used and "Diam" means that a diamond tiling was used. "XXX-perf" indicates that we consider the variant of the kernel with a perfect loop nest.

VI. Related Work

Loop transformation languages: Classic loop transformations are already specified as directives for OpenMP [24], Clang/LLVM [23], or OpenACC [6] in the context of kernel offloading. Script languages [12] and DSLs [30] were also proposed. Some of them ease the composition of transformations [12], [30], but none of them express general affine transformations, nor general loop tiling, required to exploit the full potential of polyhedral transformations.

Verification of program transformations: Several works focus on verifying that a transformed program is still equivalent to the original one. A first option is to formally prove that the compiler transformation used are correct by construction [26], [27]. Another option consists of proving the equivalence between the original program and the transformed program [3], [20], [32], [36]. Other approaches focus on a specific property, such as the preservation of the dependences of the program. Our contribution falls into this category. Polycheck [2] dynamically checks the memory accesses and ensure that they are performed in the same order than the original code. In contrast, we statically prove the correctness of the program transformation itself. In the context of automatic correction of loop transformations, [34] checks the correctness of an affine schedule using the affine form of farkas lemma. Though this method performs better than a general SAT checking, they cannot handled parametric tiling. Also, the method used to remove redondant dependences is somehow different to ours, as we want dependence functions, not general relations.

Tiled code generation: When the tile sizes are constants or depends on a single scaling parameter (monoparametric) then the tiled program can be expressed inside the polyhedral model, and a polyhedral code generator [4] can be used to generate it. When the tile sizes are unknown during the compilation, then the transformed program is not polyhedral and a common solution to generate tiled code is to merge the (parametric) tiling transformation with the code generation pass [17], [21]. Another solution is to extend the polyhedral model to express the tiled program [1], [16].

VII. Conclusion

In this paper, we have proposed a pragma language to specify a polyhedral code transformation directly in the code, a verification algorithm to check the correctness of the specified transformation and a tool suite for checking and applying the program transformation. With our formalism, different kinds of loop tilings may be expressed and may coexist, including general loop tiling with an arbitrary polyhedral tile shape, which, for instance, enable the expression of monoparametric hexagonal tiling. The tile size may be parametrized by means of a scaling parameter, whose correct values may be inferred by our verification algorithm.

In the future, we plan to improve the performance of the verification algorithm. For instance, memoization may help to cut the research space. Runtime verification, coupled with static analysis, may also further improve the performance.

References

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

[2] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. Polycheck: Dynamic verification of iteration space transformations on affine programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL16, pages 539–554, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Denis Barthou, Paul Feautrier, and Xavier Redon. On the Equivalence of Two Systems of Affine Recurrence Equations (Research Note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 309–313, 2002.

[4] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *2nd International Symposium on Parallel and Distributed Computing (ISPDC 2003), 13-14 October 2003, Ljubljana, Slovenia*, pages 23–30, 2003.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113, 2008.

[6] OpenACC Non-Profit Corporation. The openacc application programming interface version 2.0. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, jun 2013.

[7] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. Corresponding software tool PIP: http://www.piplib.org/.

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

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

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[11] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

[12] Sylvain Girbal. *Optimisation d'applications, composition de transformations de programme : modèle et outils*. PhD thesis, Université de Paris XI Orsay, 2005.

[13] GNU. Gnu linear programming kit. https://www.gnu.org/software/glpk/.

[14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. 2014.

[15] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014.

[16] Armin Grosslinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In *Proceedings of the 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, pages 1–12, 2004.

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

[18] Guillaume Iooss, Christophe Alias, and Sanjay Rajopadhye. Monoparametric tiling of polyhedral programs. https://hal.inria.fr/hal-02493164/document, Feb 2020.

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

[20] Chandan Karfa, Kunal Banerjee, Dipankar Sarkar, and Chittaranjan Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32:1787–1800, 11 2013.

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

[22] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235244, June 2007.

[23] Michael Kruse and Hal Finkel. User-directed loop-transformations in clang. In *5th International Workshop on the LLVM Compiler Infrastructure in HPC*.

[24] Michael Kruse and Hal Finkel. Design and use of loop-transformation pragmas. In *15th International Workshop on OpenMP*, 2019.

[25] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGARCH Computer Architecture News*, 19(2):63–74, 1991.

[26] Xavier Leroy. The compcert C compiler. http://compcert.inria.fr/.

[27] William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In Matt Kaufmann and LawrenceC. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 371–386. Springer Berlin Heidelberg, 2010.

[28] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

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

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'13, pages 519–530, New York, NY, USA, 2013. ACM.

[31] Sanjay V. Rajopadhye, S. Purushothaman, and Richard M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 241 of *Lecture Notes in Computer Science*, pages 488–503. Springer Berlin Heidelberg, 1986.

[32] Markus Schordan, Pei-Hung Lin, Dan Quinlan, and Louis-Noël Pouchet. Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*. 2014.

[33] Field G. Van Zee and Robert A. Van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.*, 41(3), June 2015.

[34] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS 06, page 335344, New York, NY, USA, 2006. Association for Computing Machinery.

[35] Sven Verdoolaege. ISL: An integer set library for the polyhedral model. In *ICMS*, volume 6327, pages 299–302. Springer, 2010.

[36] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. *ACM Transactions on Programming Languages and Systems*, 34(3):11:1–11:35, November 2012.