On the Verification of Polyhedral Program Transformations
Christophe Alias, Guillaume Iooss, Sanjay Rajopadhye

To cite this version:

HAL Id: hal-03106070
https://hal.science/hal-03106070
Submitted on 11 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.
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, our 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-parallelepiped), 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 tile 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 (MPPCHECK1) and code generation (MPPCODEGEN2) 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.

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

...
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 \((S,i)\) would overlap with the first parallel iterations \((T,i,0)\), 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]=y[i]+a[i][j]*x[j])  
When  
  | -1+N >= 0  
  | -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:  
#0 --[FLOW]--> #1, read #1, depth 1  
y[i]=y[i]+a[i][j]*x[j])  
When  
  | -2+N >= 0  
  | -1+N >= 0  
#0 i=0 j=0 tile_counter_0=0 tile_counter_1=0
```

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

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][1-3][6]
  #pragma tile_shape_closed[i-t][t+1][t+1]
  #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
Hence, the second outer tile counter (with general tiling shapes, and non-tiled statements. pedic tiling (and possibly different tile size/ratio), statements

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

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:

\[ C := \forall (S, \vec{i}), \langle T, \vec{j} \rangle : \langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \Rightarrow \theta_S(\vec{i}) \ll \theta_T(\vec{j}) \]  

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

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

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 \( \text{UNSAT} \neg C \), the schedule is correct. When \( \text{SAT} \neg C \), however, the schedule is not correct and we may pick a counterexample in \( \neg C \).

B. Affine tiling

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

\[ \forall (S, \vec{i}), \langle T, \vec{j} \rangle : \langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \land \]

\[ C := \exists (S, \vec{i}), \langle T, \vec{j} \rangle : \langle S, \vec{i} \rangle \rightarrow \langle T, \vec{j} \rangle \land \}

\[ \theta_S(\vec{i}_{\text{block}}, \vec{i}_{\text{local}}) \ll \theta_T(\vec{j}_{\text{block}}, \vec{j}_{\text{local}}) \]

When the tile size \( \vec{s} \) is constant, \( T \) is affine per statement, hence we may check \( \neg C \) 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 \( T \).

C. Monoparametric tiling

As soon as a tile size depends on a parameter, \( T_S \) is no longer affine. On the monoparametric parallelepipedic case we get stuck with the quadratic expression \( b.(\text{diag} \cdot \vec{r}), \vec{i}_{\text{block}} \) in \( T_S(\vec{i}_{\text{block}}, \vec{i}_{\text{local}}) = b.(\text{diag} \cdot \vec{r}), \vec{i}_{\text{block}} + \vec{i}_{\text{local}} \) (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 \( b.L \vec{i}_{\text{block}} \) in \( T_S(\vec{i}_{\text{block}}, \vec{i}_{\text{local}}) = b.L \vec{i}_{\text{block}} + \vec{i}_{\text{local}} \) (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 \( \Delta_{XY} \) 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.

E. ... and composition thereof

With that lattice, the tile origins verify:

\[
\begin{pmatrix}
  t_0 \\
  i_0
\end{pmatrix} = b.\begin{pmatrix}
  1/2 \\
  -3/2
\end{pmatrix} i_{\text{block}0} + b.\begin{pmatrix}
  0 \\
  6
\end{pmatrix} i_{\text{block}1}
\]

Hence, the second outer tile counter (\( i_{\text{block}1} \), variable tile_counter_1) will iterates through a layer of hexagons along the \( i \) axis, while the first outer counter (\( i_{\text{block}0} \), 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.

---

**Figure 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 \( b = 1 \).
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 \(\langle T, j \rangle\) to the last operation executed before \(\langle T, j \rangle\), which writes its \(k\)th read address: \(\text{source}_k((T, j)) \rightarrow \text{FLOW} \langle T, j \rangle\) for any \(j \in D_T\). 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 \(\text{source}_0((T, i, j))\) 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: \(\langle S, \bar{i} \rangle \rightarrow \text{OUTPUT} \ \text{NextWrite}(\langle S, \bar{i} \rangle)\) for any \(i \in D_S\). Finally, anti-dependences are specified by a function NextWriteForRead\(_k\) mapping an operation to the next write to the address accessed by its \(k\)th read: \(\langle S, \bar{i} \rangle \rightarrow \text{ANTI} \ \text{NextWriteForRead}_k(\langle S, \bar{i} \rangle)\) for any \(i \in D_S\) 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 and output; and two branches from \(S\) to \(T\) with type labels flow and output. All these 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 \(\delta\). Let \(\delta'\) be the smallest dependence relation such as:
- source\((S, \bar{i}) \rightarrow \text{FLOW} \langle S, \bar{i} \rangle\) for any \(\langle S, \bar{i} \rangle \in D_S\)
- \(\langle S, \bar{i} \rangle \rightarrow \text{OUTPUT} \ \text{NextWrite}(\langle S, \bar{i} \rangle)\) for any \(i \in D_S\).
- \(\langle S, \bar{i} \rangle \rightarrow \text{ANTI} \ \text{NextWriteForRead}_k(\langle S, \bar{i} \rangle)\) for any \(i \in D_S\) and any read \(k\).

Then, the transitive closure of \(\delta'\) is \(\delta\).

**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 \(\langle \delta' \rangle \subseteq \langle \delta \rangle\).
- We now prove that \(\delta \subseteq \langle \delta' \rangle\).
  - Let \(W \rightarrow \text{OUTPUT} W' \in \delta\). Since \(W \rightarrow \text{seq} W', W' = \text{NextWrite}^p(W)\) for some \(p\). Hence \(W \rightarrow \text{OUTPUT} W' \in \langle \delta' \rangle\).
  - Let \(W \rightarrow \text{FLOW} R \in \delta\). If \(W = \text{source}_k R\) for some read \(k\), \(W \rightarrow \text{FLOW} R \in \langle \delta' \rangle\).
    - If \(W \neq \text{source}_k R\) for any read \(k\), \(W_0\) be the next source for \(R\) whose address is written by \(W\). \(W \rightarrow \text{seq} W_0\), otherwise \(W_0\) would not be a source for write\((W)\). Then: \(W \rightarrow \text{OUTPUT} W_0 \in \delta\).
      - Hence the chain: \(W \rightarrow \text{OUTPUT} W_0 \rightarrow \text{FLOW} W_0 \rightarrow \text{FLOW} R_0 \rightarrow \text{FLOW} W_0 \rightarrow \text{FLOW} R_ο \rightarrow \text{FLOW} W_0 \rightarrow \text{FLOW} R\).
    - Thus \(W \rightarrow \text{FLOW} R \in \langle \delta' \rangle\).
  - Let \(R \rightarrow \text{ANTI} W \in \delta\).
    - Let \(W_0\) be the next write for the read of \(R\) whose address is written by \(W\).
      - If \(W = W_0\), then \(R \rightarrow \text{ANTI} W \in \delta' \subseteq \langle \delta' \rangle\).
      - If \(W \neq W_0\), then \(W \rightarrow \text{seq} W_0\), otherwise \(W_0\) would not be the next write. Hence \(W \rightarrow \text{OUTPUT} W \in \delta\).
    - Hence the chain: \(R \rightarrow \text{ANTI} W_0 \rightarrow \text{OUTPUT} W \rightarrow \text{FLOW} R \rightarrow \text{FLOW} W_0 \rightarrow \text{FLOW} R\).
    - Thus: \(R \rightarrow \text{ANTI} W \in \langle \delta' \rangle\).

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 \(\langle S, \bar{i}(\bar{j}) \rangle \rightarrow \text{FLOW} \langle T, \bar{j} \rangle\) when \(\bar{j} \in D\). If \(T_S(\bar{i}_{\text{block}}, \bar{i}_{\text{local}}) = \bar{i}\) and \(\bar{T}(\bar{j}_{\text{block}}, \bar{j}_{\text{local}}) = \bar{j}\), then the version of \(u\) operating in the tiled domain is \(\bar{u} = T_S^{-1} \circ u \circ T_T\) and the tiled guard is \(\bar{D} = T_T^{-1} (\bar{D})\). The tiled dependence branch \(\langle S, \bar{i}(\bar{j}) \rangle \rightarrow \text{FLOW} \langle T, \bar{j} \rangle\) when \(\bar{j} \in D \setminus \bar{D}\). We tile anti and output dependence in the same fashion.

We already proposed an algorithm, in the preprint [18], to tile a polyhedron \(\bar{D} \mapsto \bar{D}\) and an affine function \(u \mapsto \bar{u}\). We also proved the polyhedral closure of parallelepipedic and general monoparametric tiling: \(\bar{D}\) 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, $D_S$ is the set of iterations $(j_{\text{block}}, j_{\text{local}}, i_{\text{local}})$ satisfying:

\[
[j_{\text{block}} = j_{\text{local}} = 0 \land i_{\text{block}} = N_{\text{block}} \land 0 \leq i_{\text{local}} < N_{\text{local}}] \lor \\
[j_{\text{block}} = j_{\text{local}} = 0 \land 0 \leq i_{\text{block}} < N_{\text{block}} \land 0 \leq i_{\text{local}} < b]
\]

where $N = b N_{\text{block}} + N_{\text{local}}, 0 \leq N_{\text{local}} < b$. 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). $D_T$ 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 $\neg C$. When the schedule is wrong, we project $\neg C$ on $b$ to get the domain of bad values for the scaling parameter, then we deduce the good ones by a simple subtraction from $\{b | b \geq 1\}$.

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 tiled code generator in a separate tool, MPPCODEGEN\(^4\). MPPCODEGEN computes the tiling from the specification and uses isscc’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:
  \[
  L = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \\
  \mathcal{P} = \{(t, i) | t + i \geq 0, t - i \geq 0, 2 - t - i > 0, 2 + i - t > 0\}.
  \]

- **j1d-perf** Hex is the same jacobi-1D variant with an hexagonal tiling:
  \[
  L = \begin{pmatrix} 1 & 0 \\ -3 & 6 \end{pmatrix} \\
  \mathcal{P} = \{(t, i) | i - t \geq 0, t + 1 \geq 0, t + i \geq 0, t + j \geq 0, -i + t + 4 > 0, 1 - t > 0, 4 - t - i > 0, 1 - t - j > 0\}.
  \]

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, $\sum_S \text{card } D_S$ (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 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 $S \rightarrow T$, the enumeration of all the tuples (source polyhedron $\in D_S$, dependence branch of $\hat{u}$, target polyhedron $\in D_T$). 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.
<table>
<thead>
<tr>
<th>Kernel</th>
<th>Build time</th>
<th>Domains (base)</th>
<th>Domains (tiled)</th>
<th>Build time</th>
<th>Build time</th>
<th>Build time</th>
<th>Checking total time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>0.09</td>
<td>2</td>
<td>12</td>
<td>0.04</td>
<td>6</td>
<td>0.04</td>
<td>0.11</td>
<td>1.64</td>
</tr>
<tr>
<td>gemver</td>
<td>0.04</td>
<td>4</td>
<td>14</td>
<td>0.05</td>
<td>12</td>
<td>0.04</td>
<td>0.05</td>
<td>0.48</td>
</tr>
<tr>
<td>gesummv</td>
<td>0.04</td>
<td>5</td>
<td>14</td>
<td>0.04</td>
<td>14</td>
<td>0.04</td>
<td>0.05</td>
<td>0.38</td>
</tr>
<tr>
<td>symm</td>
<td>0.12</td>
<td>4</td>
<td>24</td>
<td>0.09</td>
<td>13</td>
<td>0.16</td>
<td>0.25</td>
<td>1.40</td>
</tr>
<tr>
<td>syr2k</td>
<td>0.06</td>
<td>2</td>
<td>12</td>
<td>0.04</td>
<td>6</td>
<td>0.04</td>
<td>0.08</td>
<td>0.79</td>
</tr>
<tr>
<td>syrk</td>
<td>0.06</td>
<td>2</td>
<td>12</td>
<td>0.04</td>
<td>6</td>
<td>0.04</td>
<td>0.08</td>
<td>0.74</td>
</tr>
<tr>
<td>trmm</td>
<td>0.06</td>
<td>2</td>
<td>12</td>
<td>0.05</td>
<td>8</td>
<td>0.05</td>
<td>0.14</td>
<td>0.76</td>
</tr>
<tr>
<td>2mm</td>
<td>0.23</td>
<td>4</td>
<td>36</td>
<td>0.09</td>
<td>13</td>
<td>0.10</td>
<td>0.46</td>
<td>1.93</td>
</tr>
<tr>
<td>3mm</td>
<td>0.68</td>
<td>6</td>
<td>78</td>
<td>0.14</td>
<td>19</td>
<td>0.17</td>
<td>2.71</td>
<td>1.24</td>
</tr>
<tr>
<td>atax</td>
<td>0.05</td>
<td>4</td>
<td>18</td>
<td>0.03</td>
<td>11</td>
<td>0.05</td>
<td>0.11</td>
<td>1.36</td>
</tr>
<tr>
<td>bicc</td>
<td>0.03</td>
<td>4</td>
<td>12</td>
<td>0.03</td>
<td>10</td>
<td>0.04</td>
<td>0.06</td>
<td>1.49</td>
</tr>
<tr>
<td>dotgen</td>
<td>0.43</td>
<td>3</td>
<td>32</td>
<td>0.10</td>
<td>7</td>
<td>0.04</td>
<td>0.18</td>
<td>4.29</td>
</tr>
<tr>
<td>mvt</td>
<td>0.02</td>
<td>2</td>
<td>8</td>
<td>0.02</td>
<td>6</td>
<td>0.03</td>
<td>0.03</td>
<td>0.24</td>
</tr>
<tr>
<td>cholesky</td>
<td>0.10</td>
<td>4</td>
<td>18</td>
<td>0.19</td>
<td>23</td>
<td>0.95</td>
<td>0.57</td>
<td>0.61</td>
</tr>
<tr>
<td>gramschmidt</td>
<td>0.18</td>
<td>7</td>
<td>32</td>
<td>0.15</td>
<td>24</td>
<td>0.22</td>
<td>0.27</td>
<td>2.08</td>
</tr>
<tr>
<td>lu</td>
<td>0.07</td>
<td>3</td>
<td>20</td>
<td>0.20</td>
<td>18</td>
<td>0.81</td>
<td>0.56</td>
<td>0.78</td>
</tr>
<tr>
<td>trisolv</td>
<td>0.02</td>
<td>3</td>
<td>8</td>
<td>0.03</td>
<td>14</td>
<td>0.11</td>
<td>0.10</td>
<td>0.27</td>
</tr>
<tr>
<td>correlation</td>
<td>0.34</td>
<td>15</td>
<td>48</td>
<td>0.32</td>
<td>45</td>
<td>0.33</td>
<td>0.34</td>
<td>3.66</td>
</tr>
<tr>
<td>covariance</td>
<td>0.18</td>
<td>8</td>
<td>32</td>
<td>0.23</td>
<td>35</td>
<td>0.38</td>
<td>0.35</td>
<td>1.47</td>
</tr>
<tr>
<td>floyd-warshall</td>
<td>0.04</td>
<td>1</td>
<td>8</td>
<td>0.07</td>
<td>23</td>
<td>0.23</td>
<td>0.54</td>
<td>1.60</td>
</tr>
<tr>
<td>ftd-2d</td>
<td>1.51</td>
<td>4</td>
<td>160</td>
<td>0.18</td>
<td>27</td>
<td>2.27</td>
<td>9.63</td>
<td>11.18</td>
</tr>
<tr>
<td>heat-3d</td>
<td>9'23</td>
<td>2</td>
<td>940</td>
<td>0.81</td>
<td>42</td>
<td>4.57</td>
<td>3'1</td>
<td>22'22</td>
</tr>
<tr>
<td>h3d-perf Hex</td>
<td>45'2</td>
<td>2</td>
<td>1154</td>
<td>0.56</td>
<td>10</td>
<td>39.7</td>
<td>4h4'6</td>
<td>5h29'49</td>
</tr>
<tr>
<td>jacob-1d</td>
<td>0.15</td>
<td>2</td>
<td>42</td>
<td>0.04</td>
<td>14</td>
<td>0.14</td>
<td>0.82</td>
<td>1.69</td>
</tr>
<tr>
<td>j1d-perf Diam</td>
<td>0.83</td>
<td>1</td>
<td>37</td>
<td>0.02</td>
<td>3</td>
<td>0.10</td>
<td>1.72</td>
<td>25.39</td>
</tr>
<tr>
<td>j1d-perf Hex</td>
<td>1.28</td>
<td>1</td>
<td>54</td>
<td>0.02</td>
<td>3</td>
<td>0.21</td>
<td>2.41</td>
<td>44.34</td>
</tr>
<tr>
<td>jacob-2d</td>
<td>3.75</td>
<td>2</td>
<td>134</td>
<td>0.14</td>
<td>22</td>
<td>0.50</td>
<td>8.94</td>
<td>16.59</td>
</tr>
<tr>
<td>j2d-perf Hex</td>
<td>41.70</td>
<td>1</td>
<td>273</td>
<td>0.07</td>
<td>5</td>
<td>1.24</td>
<td>2'2</td>
<td>4'10</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>1.31</td>
<td>1</td>
<td>74</td>
<td>0.14</td>
<td>19</td>
<td>1.47</td>
<td>13.67</td>
<td>4.56</td>
</tr>
</tbody>
</table>

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 redundant 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


