Scalable Polyhedral Compilation, Syntax vs. Semantics: 1–0 in the First Round

Riyadh Baghdadi
baghdadi@mit.edu
Massachusetts Institute of Technology

Albert Cohen
Google
albertcohen@google.com

Abstract

The development of lightweight polyhedral compilation algorithms opens polyhedral loop transformation, parallelization and code generation to a larger class or programs. The Pluto scheduling algorithm plays a major role in state-of-the-art polyhedral compilers, aiming for the simultaneous enhancement of locality and the exploitation of coarse-grain parallelism through loop tiling. Reducing the run time of affine scheduling algorithms like Pluto has a significant impact on the overall compilation time of polyhedral compilers. Several approaches have been proposed to reduce the run time of affine scheduling while preserving most of the optimization opportunities. Yet these works have taken separate rather than consolidated attempts at the problem. In an attempt to better characterize the potential and limitations of such approaches, we introduce and evaluate a family of techniques called *offline statement clustering*. Program statements are clustered into macro-statements and the dependence graph is projected onto these macro-statements before affine scheduling. Offline statement clustering integrates transparently into the flow of a state-of-the-art polyhedral compiler and can reduce the scheduling time by a factor of 6 (median) without inducing a significant loss in optimization opportunities. We also study the theoretical and experimental properties of statement clustering, shedding new light on the leading syntax-driven heuristic [14]. Our work-in-progress study confirms the surprising finding that the simpler, apparently more fragile and syntax-dependent methods tend to perform well on a wide range of benchmarks.

1 Introduction

The Pluto algorithm [7] is widely used for affine scheduling. It reorders (schedules) the instances of statements in order to enhance the locality of memory accesses and expose parallelism. Reducing the execution time of this algorithm is an important step towards reducing the overall compilation time in many polyhedral compilers. Indeed, besides the Pluto source-to-source compiler, the algorithm has been adapted into Graphite [17, 21] and Polly [11], the polyhedral compilation passes of GCC and LLVM, respectively, as well as Tiramisu [6] and `PPCG` [2–4, 23] source-to-source compilers.

To find a new schedule of a group of statements in a loop nest, the Pluto algorithm proceeds level by level, progressing inwards starting from the outermost level. The constructed schedule is multidimensional (a multidimensional schedule defines the logical time of execution of a statement instance compared to the other statement instances). The Pluto algorithm computes one dimension of the multidimensional schedule of each one of the program statements at each loop level.

Our work originates from the following observation: at a given loop level, when the Pluto algorithm looks for a schedule, i.e. when it looks for a new execution order statement instances, it is very likely to assign the same schedule to the statements that are a part of the same strongly connected component (SCC) in the dependence graph at that loop level. If the schedules of some statements are identical, or almost equal, a natural question is to ask whether one can take advantage of this observation in order to accelerate affine scheduling. In fact, there is no point in computing a separate schedule for each one of these statements. It would be less expensive to represent all of them as a single *macro-statement* and only schedule that macro-statement. This naturally leads to *statement clustering*.

In statement clustering, the polyhedral representation of a program is transformed into a new one where groups of statements are clustered into macro-statements. Deciding which statements should be clustered is done through a clustering heuristic. Then, instead of scheduling the original statements of the program, we schedule the macro-statements.

Assuming that D is the set of the program statements, the Pluto affine scheduling algorithm is estimated to have an average complexity of $\mathcal{O}|D|^5$ [22], i.e., a power five complexity in the number of statements on average. Since the number of macro-statements is lower than or

IMPACT 2020, January 22, 2020, Bologna, Italy

equal to the number of the original statements of the program, it is expected that scheduling macro-statements may take less time than scheduling the original statements. This can be explained in more detail as follows: the Pluto affine scheduling algorithm relies on integer linear programming (ILP) and on an ILP solver. To find a new schedule for the program statements, it creates an ILP system of constraints and solves that system. The number of variables in the ILP system of constraints is proportional to the number of statements and the number of constraints is roughly proportional to the number of dependences of the program [7]. Using statement clustering, we create a new representation of the program that has fewer statements and fewer dependences, which reduces the total number of variables and the total number of constraints in the ILP problem. It is expected that the new simplified system of constraints, which is smaller that the original system of constraints, is more likely to be solved more quickly by the ILP solver. Note that this is not automatic however. It is well known that the practical complexity of ILP is not easily correlated to the size of the input problem. Indeed, although reducing the size of the input of the scheduling algorithm reduces the scheduling time in general (as we show experimentally), there is no theoretical guarantee about this. The reason is that the time needed by the ILP solver to find new schedules does not depend only on the number of variables and on the number of constraints in the constraint system but depends also on other factors, such as the sparsity of the constraint matrix and the absolute values of the coefficients. All in all, it is difficult to validate the effectiveness of clustering approaches without a careful experimental evaluation, considering different optimization objectives and diverse benchmarks.

Independently, several other works considered another, more syntax-directed approach to statement clustering (although they do not call it statement clustering). From their original inception, the Graphite pass in GCC [17, 21] and the Polly pass in LLVM [11] have been considering basic blocks of the compiler's intermediate representation as macro-statements. Mehta and Yew [14] proposed an algorithm to project a statement-wise dependence graph onto these macro-statements. They presented the first systematic evaluation of the impact of basic block clustering, and successfully scheduled larger benchmarks of unprecedented size from the NAS and SPEC CPU suites. Yet this syntax-directed approach remains controversial among compiler experts, for its intrinsic fragility, and because no general framework has been proposed to compare its effectiveness and robustness with alternative methods. Moreover, the previous approaches are not designed to decouple the exploration of the clustering heuristics from the correctness and the transformation. Decoupling these heuristics and allowing for a fair comparison is the main purpose of our work: starting from the SCC-based observation, we propose a general framework for statement clustering that is independent from the clustering heuristic, and we use it to evaluate two clustering heuristics.

Note that we only focus on reducing the time for the step of affine scheduling. Reducing the time of the other steps is not in the scope of this work. Note also that we do not change the complexity of the Pluto affine scheduling algorithm, we rather suggest a practical technique to reduce its execution time.

As a summary, we introduce a general framework to improve the scalability of affine scheduling algorithms called *offline statement clustering*. The contributions of this paper are the following:

- We present a general framework for statement clustering. We make a clear distinction between the correctness of statement clustering and heuristics to decide which statements should be grouped into a single macro-statement for affine scheduling.
- We present two clustering heuristics: the "syntactic" basic-block (BB) heuristic and the "semantic" strongly-connected-component (SCC) heuristic. These heuristics derive from different observations about the likelihood of individual program statements to be assigned an identical affine schedule. While the semantic heuristic comes with theoretical guarantees about the preservation of profitable affine transformations, it is more complex and is slightly less aggressive at simplifying the scheduling problem than the syntactic one.
- Our experiments show that statement clustering reduces affine scheduling time by a factor of 6 (median) without a significant loss in optimization opportunities.
- Despite its fragile ground and sensitivity to syntactic choices in the source program, basic block clustering is remarkably effective at improving compilation time without impacting the performance of the generated code. This last result provides an independent cross-validation of the results of Mehta and Yew. This is also confirmed, surprisingly, when applying loop-distribution affine scheduling heuristics, an option Mehta and Yew did not consider.

Still, a syntactic technique cannot be a long-term answer to such an essential problem for scalable polyhedral compilation. Our work-in-progress results show that the study of more robust clustering methods is not going to be an easy one with low-hanging fruits in compilation or application execution time. The paper concludes with the sketch of one promising direction.

2 Background and Notations

A macro-statement is a virtual statement representing a cluster of statements. A macro-statement is represented using an iteration domain. We say that a statement S corresponds to a macro-statement M if the macro-statement M was created by clustering S with zero or more statements in the same macro-statement.

A clustering heuristic decides which statements should be grouped together. The output of the heuristic algorithm is a *clustering decision*. It is a set of clusters, where each cluster represents a macro-statement. Each cluster is a set of iteration domains (i.e., a set of statements).

3 Example of Statement Clustering

Let us introduce statement clustering on the simple example in Figure 1.

```
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
S1  temp1 = A[i][j] * B[i][j];
S2  C[i][j] = temp1;

S3  temp2 = A[i][j] * C[i][j];
S4  D[i][j] = temp2;
    }
```

Figure 1. Illustrative example for statement clustering.

This program has 4 statements and 7 dependence relations. Figure 2 shows the skeleton of the dependence graph, without polyhedral annotations on the edges. To apply statement clustering one first needs to select a clustering heuristic. Let us suppose that the heuristic suggests the following clustering decision:

$$\{\{D_{S_1}; D_{S_2}\}; \{D_{S_3}; D_{S_4}\}\}$$

This clustering decision indicates that the two statement domains D_{S_1} and D_{S_2} should be clustered together and that the two statement domains D_{S_3} and D_{S_4} should be clustered together.

Applying this clustering decision yields new iteration domains and a new dependence graph for the program. The skeleton of the new dependence graph is presented in Figure 3. Vertices are the macro-statements and edges are the projection of the original dependences onto these macro-statements. The clustered dependence graph has 3 dependence relations and 2 macro-statements: M_1 which represents the clustering of D_{S_1} and D_{S_2} and M_2 which represents the clustering of D_{S_3} and D_{S_4}.

4 Clustering Framework

The goal of a clustering algorithm is to construct a set of macro-statements and a dependence graph between these macro-statements. For now, we are not interested in deciding which statements should be clustered together, such a decision will be taken by a separate heuristic and will be the subject of Section 6. We restrict ourselves

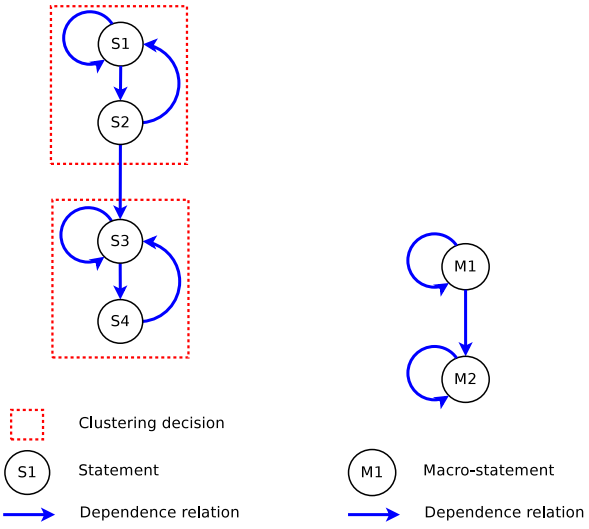

Figure 2. Original dependence graph.

Figure 3. Dependence graph after clustering.

to clustering algorithms that group statements that are a part of the same basic-block, i.e., that do not cluster across basic blocks. The study of Mehta and Yew and our study of the *swim* benchmark shows that this already opens to large factors of reduction in the size of the dependence graphs. A more generic clustering approach considering statements that are not a part of the same basic-block is left for future work.

A clustering algorithm takes as input (1) the set of statement iteration domains of the original program; (2) the graph of dependences between these statements; and (3) a clustering decision: a set of clusters of statements (where each cluster is a set of iteration domains).

A clustering algorithm returns (1) the set of macro-statement domains; and (2) the graph of dependences between the macro-statements.

Since we consider clusters of statements within the same basic block, we assume that all statements that are clustered together have the same number enclosing loops, or dimensions.

Under these hypotheses, offline statement clustering obeys the general structure of Algorithm 1.

1. First, create a mapping, M from the iteration domain of each statement to the iteration domain of the corresponding macro-statement. This mapping will be used to transform the statements into macro-statements. It is created as follows: for each cluster in the clustering decision, generate a new name for the macro-statement (let us assume that N is the generated name), for each statement in the cluster, create a map from the iteration domain of that statement into a new iteration domain that has the same dimensions but has N as a name.

This is the iteration domain of the corresponding macro-statement.

2. In the second step, create the set D' of iteration domains of the macro-statements. This is done by applying the mapping M to the iteration domains of the program statements. D' is the union of all the resulting iteration domains.
3. In the third step, create the set Δ' of dependences between the macro-statements. This is done by applying the mapping M to each one of the dependences of the program. Δ' is the union of all the resulting dependences. The number of dependences in Δ' is less than the number of dependences in the original dependence graph because after transforming the original dependences of the program, we may get some redundant dependences.

```
Input:
D: a set of the iteration domains of the program statements.
Δ: a set of the dependences between the program statements.
ClusteringDecision: a set of sets of iteration domains.
Output:
D': a set of the macro-statement iteration domains.
Δ': a set of the dependences between the macro-statements.
// Step 1: create a mapping
foreach cluster ∈ ClusteringDecision do
    // Generate a name (identifier) for the new
       macro-statement
    N ← GenerateName();
    M ← {} ;
    foreach D_S ∈ cluster do
        // We assume that S is the name of the statement
           represented by D_S, and n is the number of
           dimensions of D_S
        // Create a map M that maps the iteration domain
           of S into a new iteration domain
           (macro-statement's iteration domain)
        m ← {Si_1, ..., i_n → Ni_1, ..., i_n};
        M ← M ∪ m ;
    end
end
// Step 2: create macro-statements
foreach cluster ∈ ClusteringDecision do
    D' ← {} ;
    foreach D_S ∈ cluster do
        // apply M to D_S
        D'_S ← M D_S;
        D' ← D' ∪ D'_S ;
    end
end
// Step 3: transform Δ from a graph of dependences
   between statements into a graph of dependences between
   macro-statements
Δ' ← {} ;
foreach δ ∈ Δ do
    δ' ← apply M to the source and sink of δ; // this
       returns a map from MSourceδ to MSinkδ
    Δ' ← Δ' ∪ δ';
end
```

Algorithm 1: Clustering algorithm template.

5 Applying Offline Clustering to the Pluto Affine Scheduling Algorithm

No modification of the Pluto affine scheduling algorithm is needed to take advantage of offline clustering. One only needs to apply offline clustering before solving the affine scheduling problem:

1. Apply the clustering heuristic. This heuristic creates a clustering decision. The output of this step is passed as an argument to the clustering algorithm.
2. Perform offline clustering to create a new dependence graph with a set of macro-statements.
3. Apply the Pluto affine scheduling algorithm on the macro-statements and clustered dependence graph. This results in a new schedule for the macro-statements.
4. Deduce the schedule of the original statements of the program as follows:
   - For each macro-statement M:
     - For each statement S that corresponds to M, the schedule of S is equal to the schedule of M except that a static dimension is added to the schedule of S. This static dimension is used to order the statements of the same macro-statement among themselves lexicographically in the same basic-block according to their original lexicographical order (or in any order that satisfies the dependences between these statements).
5. The original representation along with the newly computed schedule of the program are used in the rest of the polyhedral compilation flow.

Correctness of Transformations after Clustering A loop transformation is correct if it does not violate the execution order imposed by the dependences of the program. Let us assume that a statement S_1 is part of a macro-statement M_1 and that the statement S_1 is the source (or the sink) of a dependence δ in the original program dependence graph. When clustering, the source of the dependence δ (or its sink) becomes M_1 and the schedule of M_1 will be constrained by the dependence δ which means that the schedules of all the statements that correspond to M_1 are constrained also by the dependence δ. This means that statement clustering, in fact, only adds additional constraints to the schedules but does not remove any existing constraint. In any case, a statement is never going to be less constrained than it should be and thus a correct transformation is still assured after statement clustering.

6 Clustering Heuristics

The goal of a clustering heuristic is to decide which statements should be clustered together in the same macro-statement. The clustering heuristic only clusters statements that belong to the same basic-block together and thus it needs to know which statement belongs to which basic-block. This information is not encoded in

the iteration domains of the statements but is extracted from the original AST of the program.

6.1 Clustering Decision Validity

Clustering a given set of statements together is not always valid. In this section we show when is it valid to cluster a given set of statements in the same macro-statement. Although we only consider the case where statements are a part of the same basic-block, the validity criterion is defined for the general case (i.e., when statements to be clustered together are not necessarily in the same basic-block).

To reason about the correctness of a given clustering, let us consider the following example:

```
for (i = 0; i < N; i++)
S1 A[i] = B[i] + 1;

for (i = 0; i < N; i++)
S2 C[i] = A[i] + C[i - 1];

for (i = 0; i < N; i++)
S3 D[i] = A[i] + C[i];
```

Figure 4 shows the flow dependences between the statements S_1, S_2 and S_3. If the two statements S_1 and S_3 are clustered together, there are two possibilities regarding the order of execution of S_2 with regard to the two statements S_1 and S_3: S_2 may be scheduled to be executed before the two statements or after them. Either ordering violates a dependence and thus clustering S_1 and S_3 is not valid in this case.

In general, given a dependence graph, it is not possible to cluster two statements if this would yield a cycle in the dependence graph or if the resulting cluster is not convex (i.e., a path of dependences between these two statements passes by a third one that does not belong to the cluster.

This criterion always needs to be verified to check whether a given clustering decision is valid unless the clustering heuristic is guaranteed always to produce valid clustering decisions (such as the SCC clustering heuristic).

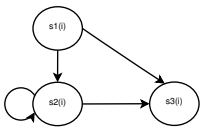

Figure 4. Flow dependence graph.

In the following two sections we present examples of clustering heuristics.

6.2 SCC Clustering

The SCC clustering heuristic relies on analyzing the properties of the dependence graph of the program (which represents the semantics of the program). The SCC clustering heuristic proceeds as follows: for each loop level k in a given loop nest (k is between 1 and the maximal depth of the loop), we restrict the dependence graph on the statements of that loop nest and on the loop level k. Then, we compute the strongly connected components of the restricted dependence graph. If a set of statements are strongly connected together at all the loop levels, then the SCC clustering heuristic decides to cluster them together. The remaining of this section provides a more formal description of the SCC heuristic.

Let $\Delta_k = V, E$ be the dependence graph of the program restricted to the statements that belong to *one basic-block* (let us call it BB1) and restricted to the loop depth k (k is between 1 and the maximal depth of the loop). The vertices V of Δ_k are the statements of BB1 and the edges E are the dependence relations between these statements projected on the k dimension.

Let $\Delta'_k = V', E'$ be a dependence restricted also to the statements of BB1 and to the loop level k. This graph is the graph of dependences between the *execution instances* of the statements of BB1 (unlike Δ_k which is the graph of dependences between the *statements* of BB1). The vertices V' of Δ'_k are the execution instances of the statements of BB1 and the edges E' are the dependences between these execution instances.

Let us assume that Δ_k has an SCC that we call ω. Let V_ω be the set of vertices (statements) of this SCC. We say that ω is *dense* if for each two statements in V_ω, for each two instances of these two statements, there is a directed path in Δ'_k between these two instances.

To perform SCC based clustering, we compute Ω, the set of strongly connected components (SCCs) in the dependence graph Δ_k. For each $\omega \in \Omega$ (i.e., for each SCC in Ω), if ω is *dense*, we mark the statements of ω as candidates for clustering together at depth k. If a set of statements are marked as candidates for clustering together at all the loop depths of a given loop, then those statements are marked by the heuristic to be actually clustered together. We do the previous procedure for all the basic-blocks of the program.

Validity of SCC Clustering The SCC clustering heuristic always produces valid clustering decisions. To verify this, we need to verify that for any two statement instances SI and $S'I'$ that are a part of an SCC, any third statement instance $S''I''$ that is a part of a path that goes from SI to $S'I'$ must also be clustered with SI and $S'I'$. Indeed, this is the case because if there is a path that goes from SI to $S'I'$ that includes $S''I''$ then $S''I''$ is also a part of the SCC and thus $S''I''$ is also clustered with SI and $S'I'$.

6.3 Basic-block Clustering

In this syntactic-based clustering heuristic, all the statements that are a part of the same basic-block are clustered together in one macro-statement. We abbreviate the basic-block clustering with BB clustering.

Let us consider the example in Figure 5, applying the basic-block clustering heuristic on this code gives the following clustering: $\{\{D_{S_0}, D_{S_1}\}, \{D_{S_2}, D_{S_3}\}, \{D_{S_4}\}\}$

```
for (i=0; i<N; i++) {
  S0;
  S1;
  for (j=0; j<N; j++) {
       S2;
       S3;
       if (l>0)
         S4;
  }
}
```

Figure 5. Illustrative example for basic-block clustering.

Possible Losses in Optimization Opportunities Basic-block clustering prevents the scheduler from assigning different schedule coefficients to the statements that are a part of the same basic-block. The same schedule coefficients (for the dynamic schedule dimensions) must be assigned to all the statements of a basic-block. Loop transformations such as loop distribution, loop shifting of only some statements of the basic-block are no more possible when such clustering is applied. The next section studies the concrete impact of the different clustering methods.

7 Experiments

We implemented offline statement clustering on top of the `PPCG` source-to-source polyhedral compiler (version 0.02) [23]. We modified `PPCG` to perform offline clustering immediately after dependence analysis and before affine scheduling. The clustering step itself is implemented using a python wrapper around `isl`.

We evaluate offline statement clustering on a set of benchmarks that represent a variety of applications including linear algebra, image processing, stencils, etc. We compare performance with and without statement clustering, and considering the extreme fusion/distribution heuristics of the Pluto algorithm, *min-fuse* and *max-fuse*, as implemented in `isl`.

The benchmarks are the following:

Image processing kernels. This is a set of benchmarks that includes 7 image processing kernels (*color conversion*, *dilate*, *2D convolution*, *gaussian smoothing*, *basic histogram*, *resize* and *affine warping*).

Polybench-3AC. When a program is transformed into three address code (3AC), the number of statements increases considerably. It is interesting to assess how much statement clustering helps in such a production-like context. Since `PPCG` is a source-to-source compiler that does not convert its input code into three address form internally, and we could not find an automatic tool that converts C code into source-form 3AC, we manually converted Polybench 3.2 into 3AC. The process of converting Polybench into 3AC is described in [5].[1]

Swim benchmark. A benchmark extracted from SPEC CPU 2000 [12] rewritten in C.

Dist kernel. A kernel extracted from the Allen and Kennedy textbook [13], where it was selected to illustrate the importance of loop distribution to expose parallelism.

Machine configuration The experiments were conducted on a system with a 64 bit Intel Core i5 CPU U470 at 1.33 GHz, with 2 cores (and 4 threads) and 2 GB of RAM, running Ubuntu GNU/Linux.

We set a 10 minute maximal compilation time for `PPCG`, killing the process when a timeout occurs.

	SCC-clustering		BB-clustering	
Benchmark	macro stmts	deps	macro stmts	deps
2mm	3.5	5.86	3.5	5.86
3mm	3.33	6.33	3.33	6.33
atax	3.5	8.2	3.5	8.2
bicg	3.5	10	4.67	13.33
cholesky	1.83	1.59	1.83	1.59
doitgen	1.67	3.67	1.67	3.67
gemm	2.5	6	2.5	6
gemver	3.75	5	3.75	5
gesummv	2.2	4.43	2.2	4.43
mvt	3	7	3	7
symm	3	3.18	4	3.89
syr2k	3.5	7.5	3.5	7.5
syrk	3	5	3	5
trisolv	2	2.83	2	2.83
trmm	4	12	4	12
durbin	2.14	2.6	2.5	2.6
dynprog	1.6	1.56	1.6	1.56
gramschmidt	1.86	2.06	1.86	2.06
lu	2.5	3.5	2.5	3.5
ludcmp	1.5	1.42	1.5	1.42
correlation	1.57	2.13	1.83	2.33
fwarshall	2	4	2	4
fdtd-2d	3.5	4.6	3.5	4.6
fdtd-apml	12.5	12.1	18.75	22
jacobi-1d-imper	2	3.5	2	3.5
jacobi-2d-imper	3	5.5	3	5.5
seidel-2d	9	35	9	35
color conversion	1	1	1	1
dilate	3.33	2	3.33	2
2D convolution	1.67	1.86	1.67	1.86
gaussian smoothing	1.33	1.33	1.33	1.33
basic histogram	1.5	2.5	1.5	2.5
resize	17	64	17	64
affine warping	25	72	25	72
dist	1	1	2	1
Swim	1	1	17.33	84.44
Median	2.5	3.67	3	4
Average	3.83	8.5	4.6	11.22

Table 1. The reduction factor in the number of statements and dependences after statement clustering.

[1] Polybench-3AC is publicly available: https://github.com/rbaghdadi/polybench-3.2-3AC.

Table 1 shows the impact of offline statement clustering on the number of statements—SCC and BB clustering: A value of 4 for the *trmm* kernel (SCC-clustering) indicates that the number of program statements is divided by 4, i.e., for every 4 statements of the original program, one macro-statement is created. A high value in the table indicates an aggressive reduction in the number of statements of the original program after statement clustering.

The median number of macro-statements is 2.5× less than the original number of statements with SCC clustering, and it is 3× less than the original number of statements with BB clustering. The number of dependences is 3.67× less than the unclustered original with SCC clustering and it is 4× less with BB clustering. Statement clustering reduces the number of statements by up to 17× and 25× for *resize* and *affine warping*, respectively.

Table 1 confirms that BB clustering is more aggressive than SCC clustering since the SCC heuristic only considers clustering SCCs that are a part of the same basic-block and in general one basic-block may contain multiple SCCs. Therefore it is natural that the number of statements is further reduced with BB clustering.

Also, in both SCC and BB clustering, the clusters (i.e., the macro-statements) are identical in all the benchmarks except *bicg*, *symm*, *durbin*, *correlation*, *fdtd-apml*, *swim* and *dist*. This means that, in 28 benchmarks out of 35, the two clustering heuristics yield exactly the same result. This is because in these benchmarks all the statements of a basic-block are a part of the same SCC. Only in 6 kernels and the larger *swim* application, the clusters produced by the SCC heuristic differ from the clusters produced by the BB heuristic.

There are extreme cases. In *color conversion*, statement clustering does not reduce the original number of statements and dependences because this kernel is composed of only one statement. In *affine warping*, statement clustering reduces the number of statements from 25 statements into only 1 macro-statement and reduces the number of dependences from 72 into 1.

In *swim*, which is a complex, periodically wrapped time-iterated stencil The default `PPCG` fails to compile the *Swim* benchmark before the 10 minutes time-out. SCC clustering does not help in reducing the number of statements: there are no SCCs at inner levels, only at the outermost time iteration. BB statement clustering succeeds in reducing the number of statements by a factor of 17.33 which enables `PPCG` to compile the benchmark in 25 seconds only.

We did not quantitatively other large benchmarks for lack of time, and because we noticed the pattern detected with *swim* is a general one: the NAS and SPEC benchmarks ported by Mehta and Yew display a similar lack of interesting SCCs at the level of innermost loops. SCCs are generally limited to small isolated kernels. This tells that the applicability of SCC-based clustering is too limited in practice to be applicable to practical problems, despite its theoretical guarantees. Another, more practical "semantical" clustering heuristic remains to be invented.

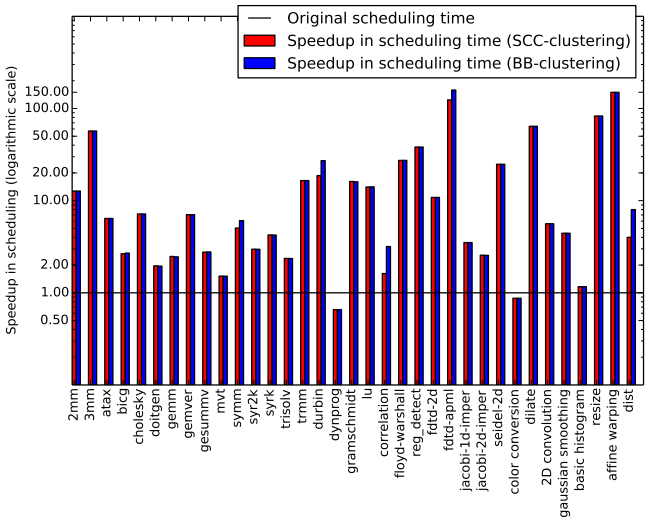

Figure 6. Improved scheduling time after clustering.

Figure 6 shows the affine scheduling time, comparing the case of statement clustering with the default `PPCG`. We take in count the overhead of the clustering algorithm, summing up the scheduling time and the clustering time. The baseline is the scheduling time without clustering: speedups higher than 1 indicate that statement clustering succeeded in reducing compilation time.

For the image processing benchmarks (Figure 6), the scheduling time for the BB and SCC heuristics are identical. This is in line with the previous results indicating that the two heuristics (SCC and BB) yield exactly the same clustering decisions in the image processing benchmarks (and also in most of the remaining benchmarks). Statement clustering for *resize* reduces the scheduling time by a factor of 82, and the reduction reaches a factor of 149 for *affine warping*. The median factor is 5.61× for all image processing benchmarks.

In the Polybench-3AC suite (Figure 6), the speedups of the BB and SCC heuristics are identical, except for 5 kernels: *symm*, *durbin*, *correlation*, *fdtd-apml* and *dist* where BB yields a higher acceleration, because BB clustering is more aggressive than SCC clustering in reducing the number of the statements in these cases.

In *dynprog*, although statement clustering reduces the number of statements by a factor of 1.6×, affine scheduling of the clustered program is slower than on the original program. This paradoxical result confirms that

the scheduling time does not depend only on the number of statements and on the number of constraints but also depends on other factors. Thus reducing the number of statements does not guarantee a reduction in the scheduling time, although it is almost always the case.

In all but one of the kernels, the benefits of statement clustering amortize the overhead of the clustering algorithm itself. The only exception is *color conversion* where statement clustering does not reduce the number of statements.

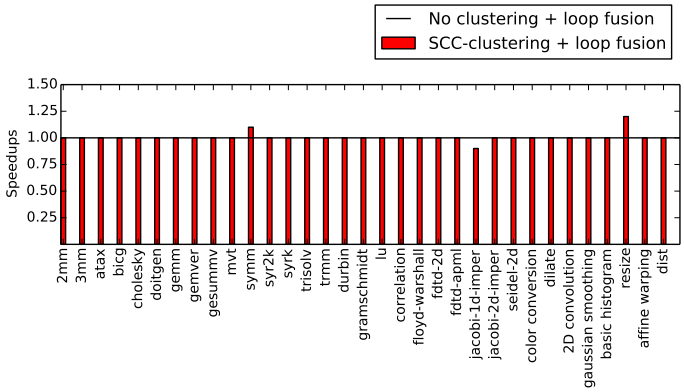

Figure 7. Improvement in execution time when SCC clustering is enabled over the case where SCC clustering is not enabled. In both cases loop fusion is enabled.

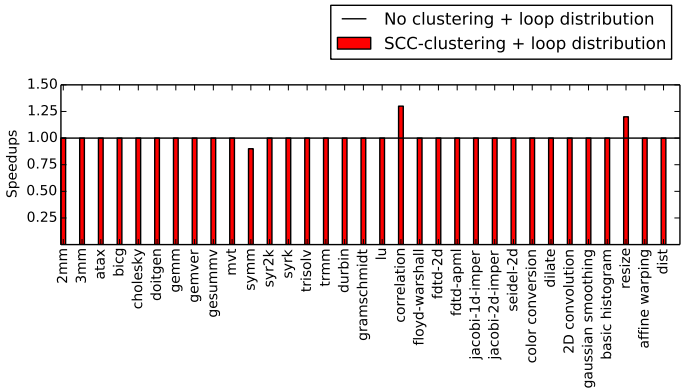

Figure 8. Improvement in execution time when SCC clustering is enabled over the case where SCC clustering is not enabled. In both cases loop distribution is enabled.

Figures 7, 8, 9 and 10 measure the effect of statement clustering on the quality of the code generated by PPCG. Figure 7 shows the speedup of the PPCG generated code when SCC statement clustering is enabled over the case where statement clustering is not enabled. In both cases, the PPCG option favoring loop fusion is enabled (`-isl-schedule-fuse=max`).

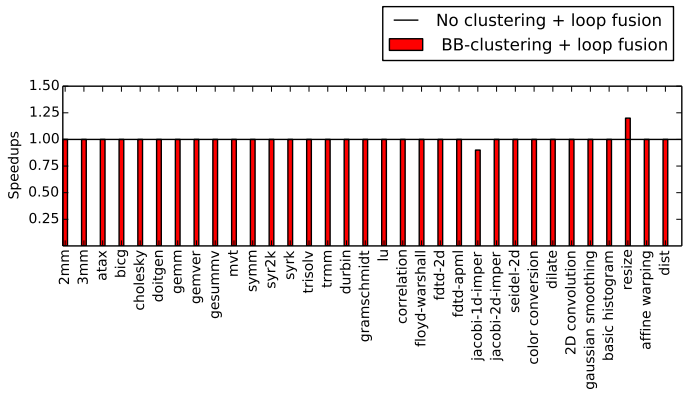

Figure 9. Improvement in execution time when BB clustering is enabled over the case where BB clustering is not enabled. In both cases loop fusion is enabled.

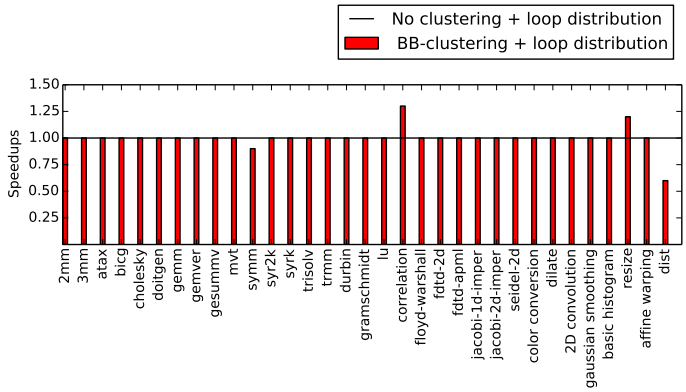

Figure 10. Improvement in execution time when BB clustering is enabled over the case where BB clustering is not enabled. In both cases loop distributed is enabled.

The goal here is to evaluate whether SCC clustering prevents PPCG from applying loop tiling and fusion. If the loops are fused or distributed when clustering is disabled but are not fused or distributed when clustering is enabled, we will notice a difference in the performance (a value different from 1 on the y axis). Values that are equal to 1 indicate that PPCG performed identically with or without clustering.

Figure 7 indicates that the performance of the code generated when statement clustering is applied is almost identical to the baseline version without clustering. In a few cases, there is a small difference but it is not due to a loss in loop tiling or fusion opportunities. In fact, we checked that PPCG with clustering could apply loop tiling and fusion in cases where PPCG without clustering can. The only difference in these benchmarks is in the order of statements within the loop body in the generated code. This difference is orthogonal to the application of statement clustering and results from choices performed by PPCG w.r.t. statement ordering in loop bodies. This ordering is arbitrary as long as it does not lead to a violation of the program dependences. In some kernels,

the statement ordering differ, inducing a small difference in the performance of the generated code.

The previous comment on Figure 7 applies also to Figures 8, 9 and 10. The only benchmark that requires attention in these experiments is *dist* in Figure 10. The code of the *dist* kernel is illustrated in Figure 11. The BB clustering heuristic with the max-fuse option yields a slowdown. This is due to the fact that BB clustering prevents loop distribution in this benchmark. Due to the loop carried dependences between the write to `A[i][j]` and the read from `A[i-1][j-1]`, the Pluto scheduler is not able to find outermost, synchronization-free parallelism. In contrast, synchronization-free parallelism can be found easily when applying loop distribution. This is possible with SCC clustering, where statements S_0 and S_1 will be clustered into one macro-statement while S_2 and S_3 will be clustered into another macro-statement. But with BB clustering, all statements will be clustered together. This is an example showing that loop distribution is actually needed to expose parallelism. SCC clustering does not prevent loop distribution while BB clustering does.

```
for (i=1; i<N; i++)
  for (i=1; i<N; i++) {
S0 t1 = B[i][j] + C[i][j];
S1 A[i][j] = t1;
S3 t2 = 2 * A[i-1][j-1];
S4 D[i][j] = t2;
  }
```

Figure 11. On the importance of loop distribution.

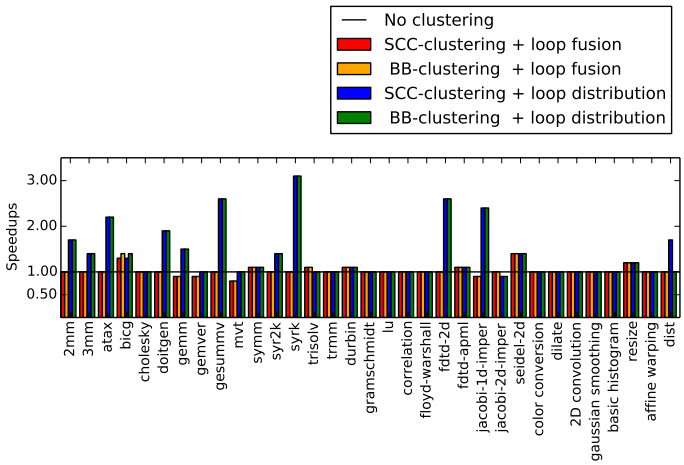

Figure 12. Improvement in execution time of SCC or BB clustering, comparing loop distribution and fusion.

Figure 12 shows the execution time of the code generated by `PPCG` when SCC and BB clustering are used along with the loop distribution/fusion-oriented options of `PPCG`. The baseline is the execution time of the `PPCG`-generated code without statement clustering. The figure shows the performance impact of loop fusion/distribution. It confirms that SCC clustering never prevents loop distribution, that both SCC and BB clustering do not prevent loop fusion, and that BB clustering occasionally prevents loop distribution (see Figures 7, 8, 9 and 10).

Although, SCC clustering is more powerful, our experiments so far confirm Mehta and Yew's finding that BB clustering should be used by default in any compiler that implements statement clustering—it is simple, effective in most case, and leaves sufficient room for applying profitable affine transformations. Our comparative study for the two clustering heuristics concludes that the simpler, syntax-based clustering heuristic tends to perform well on a wide range of benchmarks compared to the SCC clustering heuristic.

Yet this finding is only very temporary. Our study of larger applications is very partial and while it tells that SCC clustering is impractical due to the absence of interesting SCCs, it does not tell that a better semantically-grounded heuristic cannot be designed. It also does not tell that loop distribution is not important, in part because we did not conduct any array expansion (renaming, privatization) that is often necessary to expose loop distribution potential. Our work sheds light on the state of the art, with a sound basis to compare heuristics, but it does not allow to give a definite answer on statement clustering. We will hint at another direction to design a more satisfactory *online* heuristic in the conclusion.

8 Related Work

Many alternative solutions can be applied to reduce the complexity of affine scheduling. The first category of approaches has been to avoid solving a general affine scheduling problem, which generally involved integer linear programming.[2] Most scheduling algorithms in the 90s chose to address this by lowering the complexity of the dependence abstraction [8], yet the more general setting of affine dependence polyhedra, involving the affine form of the Farkas lemma has taken the upper hand as tools and heuristics matured [1, 7, 18, 19]. Other approaches focus the search on a predefined set of loop transformations [10, 20]. A more recent example is PolyMage [16], a domain-specific polyhedral compiler for a image-processing pipelines. Such approaches may gain in scalability or effectiveness, yet they are either semi-automatic or restricted to a particular context or application domain.

The Graphite pass in GCC [17, 21] and the Polly pass in LLVM [11] both consider basic blocks of the compiler's

[2] In theory many instances can be relaxed to rationals, such as Feautrier's algorithm and Pluto. Yet such relaxations induce difficult optimization and code generation challenges and have not been exploited in practice.

intermediate representation as a macro-statements. Yet these production compilers do not implement a clustering algorithm: they construct the polyhedral representation at the basic block level directly. This cannot be generalized transparently to other clustering heuristics such as SCC clustering. Moreover, this does not help conducting systematic experiments on the relevance and expressiveness-scalability trade-off of syntax-driven clustering. In effect, no Graphite- and Polly-based study investigated the the effects of basic block clustering.

In a similar way, Mehta and Yew [14] represent basic blocks as macro-statements (they call them super-statements), and they and propose an algorithm to project a statement-wise dependence graph onto these macro-statements. While their paper starts by considering multiple alternatives, they end up choosing basic block clustering as the only heuristic and do not provide a framework for comparison with other alternatives. Their experimental methodology is very systematic and establishes new records in terms of the size of the control flow regions amenable to affine scheduling. Similarly, our approach starts with a statement-wise polyhedral representation. Yet unlike their work, we propose a general setting enabling the exploration of more diverse heuristics, such as SCC clustering, with a generic criterion to establish the correctness of a given clustering decision.

Feautrier proposes the use of Communicating Regular Processes (CRP) [9]. He uses a C-like specification language in which smaller modules, comparable to functions communicate through indexed multidimensional channels. Feautrier's technique sees processes as gray boxes, exposing constraints for coarse grained scheduling of the CRP. It attempts to reduce complexity and increase scalability while preserving the existence of a global schedule. In practice the coarse grained constraints are much simpler than the original, flat scheduling problem for the full program, as coefficients and Farkas multipliers internal to the processes have been eliminated. Yet the constraints on the communication channels remain. The modules in CRP may be scheduled independently, leading to smaller size linear programs.

Most recently, Zinenko et al. [24] proposed a clustering heuristic as part of a more ambitious scheduling framework. The scheduler groups statements whose schedule dimensions can be completely aligned, while avoiding counterproductive cases where this would increase dependence distance or reduce the number of parallel dimensions. This fundamentally different approach is tightly linked with isl's scheduling algorithm, and it is not easy to reproduce it in isolation in our framework (or vice versa). It was not enabled in our experiments. Other recent clustering heuristics include the algorithm proposed by Meister et al. [15]. Like the former it adapts to the profitability and required flexibility in forming or breaking clusters according to the locality and parallelism needs. We were not aware of this technique at the time of defining our experimental methodology.

9 Conclusion and Future Work

We presented a general setting to study and compare statement clustering algorithms for the acceleration of affine scheduling. We defined a generic criterion to decide whether a given clustering decision is valid or not. We also proposed two clustering heuristics that integrate transparently with the widely-used Pluto algorithm:

- a "syntactic" heuristic similar to the block clustering of Graphite and Polly, allowing to revisiting the evaluation by Mehta and Yew [14] on a larger set of benchmarks and more diverse optimization objectives;
- a "semantic" heuristic follows the structure of the dependence graph and its SCCs.

While the latter comes with strong theoretical guarantees on the optimization space, its applicability is lower than the former in practice, due to the lack of "interesting" strongly connected components at the innermost nesting depth of the dependence graphs of real applications. The syntactic approach, owing to its simplicity and to higher compilation time improvements, appears as a very practical solution, albeit a very unsatisfactory one from a programming language stand point: it providing no guarantees on the preservation of profitable affine transformations for optimization tools.

Our evaluation considers multiple optimization objectives, with some configurations intentionally challenging the more fragile syntactic heuristic. Except for rare and synthetic cases, we confirm Mehta and Yew's finding that basic-block statement clustering is effective. It succeeds in reducing scheduling time by more than 10× in several benchmarks and by a median factor of 6× overall. And it does so without a significant loss in optimization opportunities: the performance of the unclustered code was matched for all the 33 benchmarks considered (except for the synthetic kernel highlighting the importance of loop distribution). Our experiments with PPCG illustrate the versatility of the method.

A logical next step would be to explore online clustering, an extension where it would be possible to uncluster and re-cluster statements on demand during the affine scheduling algorithm. For example, the Pluto algorithm computes schedule dimensions iteratively; if at some schedule dimension it is not able to compute a valid schedule due to statement clustering, the whole optimization would fail with offline clustering. In online clustering, it would be possible to uncluster the macro-statements at that dimension, reconsider affine transformations at a finer statement-granularity and then cluster

back. Such an approach has the potential of reconciling syntax with semantics, avoiding unrecoverable choices in offline clustering.

References

[1] Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete Modeling of Affine Transformations for Parallelism and Locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/2688500.2688512

[2] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. https://doi.org/10.1109/PACT.2015.17

[3] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven Van Haastregt, Alexey Kravets, and Alastair Donaldson. 2015. *PENCIL Language Specification*. Research Report RR-8706. INRIA. 37 pages. https://hal.inria.fr/hal-01154812

[4] Riyadh Baghdadi, Albert Cohen, Serge Guelton, Sven Verdoolaege, Jun Inoue, Tobias Grosser, Georgia Kouveli, Alexey Kravets, Anton Lokhmotov, Cedric Nugteren, Fraser Waters, and Alastair F. Donaldson. 2013. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR* abs/1302.5586 (2013). arXiv:1302.5586 http://arxiv.org/abs/1302.5586

[5] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. 2013. Improved Loop Tiling Based on the Removal of Spurious False Dependences. *ACM Trans. Archit. Code Optim.* 9, 4, Article 52 (Jan. 2013), 26 pages. https://doi.org/10.1145/2400682.2400711

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 193–205. http://dl.acm.org/citation.cfm?id=3314872.3314896

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. ACM, Tucson, AZ, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[8] Alain Darte, Yves Robert, and Frederic Vivien. 2000. *Scheduling and Automatic Parallelization* (1st ed.). Birkhauser Boston.

[9] Paul Feautrier. 2006. Scalable and structured scheduling. *International Journal of Parallel Programming* 34, 5 (2006), 459–487.

[10] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Int. J. on Parallel Programming* 34, 3 (June 2006), 261–317. Special issue on Microgrids.

[11] TOBIAS GROSSER, ARMIN GROESSLINGER, and CHRISTIAN LENGAUER. 2012. POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1142/S0129626412500107

[12] John L. Henning. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7 (July 2000), 28–35. https://doi.org/10.1109/2.869367

[13] Ken Kennedy and John R. Allen. 2002. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. http://portal.acm.org/citation.cfm?id=502981

[14] Sanyam Mehta and Pen-Chung Yew. 2015. Improving compiler scalability: optimizing large programs at small price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 143–152.

[15] Benoit Meister, Eric Papenhausen, and Benoit Pradelle. 2019. Polyhedral Tensor Schedulers. In *International Conference on High Performance Computing Simulation (HPCS)*.

[16] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 429–443. https://doi.org/10.1145/2694344.2694364

[17] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-andré Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. *In proceedings of the 2006 GCC developers summit* (2006), 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.77.9149

[18] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. 2010. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *ACM Supercomputing Conference (SC)*. New Orleans, Lousiana. 11 pages.

[19] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*. Austin, Texas.

[20] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)*. Springer-Verlag, Berlin, Heidelberg, 136–150. http://dl.acm.org/citation.cfm?id=1964536.1964546

[21] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation.

[22] Ramakrishna Upadrasta and Albert Cohen. 2013. Sub-Polyhedral Scheduling Using (Unit-)Two-Variable-Per-Inequality Polyhedra. In *ACM SIGPLAN-SIGACT Symposium on Programming Languages (POPL)*. Rome, Italy.

[23] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans.*

Archit. Code Optim. 9, 4 (Jan. 2013).

[24] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and Temporal/Spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. 3–13. https://doi.org/10.1145/3178372.3179507

A Characterization of Possible Losses in Optimization Opportunities

Without statement clustering, the scheduling algorithm is free to assign any valid schedule to the statements of the program. But with statement clustering, this is not the case anymore. The use of statement clustering implies that all the statements that are a part of the same cluster will have the same schedule (more precisely, these statements will have the same schedule coefficients for the dynamic schedule dimensions). This means that one may miss optimization opportunities. An important question is to quantify how many optimization opportunities may be lost.

Let us consider the following program:

```
for (i = 0; i < N; i++) {
S1 t = A[i];
S2 B[i] = t;
}
```

The dependence graph of this program, restricted on the two statements S_1 and S_2 at the loop level 1, has a dense SCC. Since the two statements S_1 and S_2 are part of the SCC, and since the SCC is dense, then there exists $\delta_{S_1 I \rightarrow S_2 I'}$, a dependence (or a path of dependences) from $S_1 I$ to $S_2 I'$ and there exists $\delta_{S_2 I' \rightarrow S_1 I''}$, another dependence (or a path of dependences) from $S_2 I'$ to $S_1 I''$. This is true for any $S_1 I \in D_{S_1}$, $S_2 I' \in D_{S_2}$ and $S_1 I'' \in D_{S_1}$ such that $S_1 I \ll S_2 I' \ll S_1 I''$. This means that $S_2 I'$ has to be executed between $S_1 I$ and $S_1 I''$. Any schedule for S_1 and S_2 must preserve this property.

Supposing that statement clustering is not used, one of the possible transformation that the Pluto algorithm can do in the previous example, without violating the previous property, is to shift S_2 one or more iterations as in the following example

```
for (i = 0; i < 2*N; i++)
   if (i%2 == 0)
S1   t = A[i];
   else
S2   B[i] = t;
```

Such an optimization is not possible if statement clustering is applied. In fact, the only optimization opportunity that we may lose if we apply SCC clustering is the ability to assign schedule coefficients to some of the statements that are a part of the SCC without assigning the same coefficients to all the other statements (for example, shifting one statement without shifting the others). In practice, this is not a harmful restriction as we show in Section 7.

Note that, when we apply SCC clustering, we do not lose the ability to distribute statements into separate loops, since it is not possible to distribute statements that are a part of the same SCC into two separate loops anyway [13].