Polyhedral Code Generation in the Real World

Nicolas Vasilache, Cédric Bastoul, and Albert Cohen

ALCHEMY Group, INRIA Futurs and LRI, Université Paris-Sud XI

Abstract. The polyhedral model is known to be a powerful framework to reason about high level loop transformations. Recent developments in optimizing compilers broke some generally accepted ideas about the limitations of this model. First, thanks to advances in dependence analysis for irregular access patterns, its applicability which was supposed to be limited to very simple loop nests has been extended to wide code regions. Then, new algorithms made it possible to compute the target code for hundreds of statements while this code generation step was expected not to be scalable. Such theoretical advances and new software tools allowed actors from both academia and industry to study more complex and realistic cases. Unfortunately, despite strong optimization potential of a given transformation for e.g., parallelism or data locality, code generation may still be challenging or result in high control overhead. This paper presents scalable code generation methods that make possible the application of increasingly complex program transformations. By studying the transformations themselves, we show how it is possible to benefit from their properties to dramatically improve both code generation quality and space/time complexity, with respect to the best state-of-the-art code generation tool. In addition, we build on these improvements to present a new algorithm improving generated code performance for strided domains and reindexed schedules.

1 Introduction

Compiler performance has long been quantified through the number of processed code lines per time unit. Compile time used to be (almost) linear in the code length. In order to find the best possible optimizations, present day compilers must rely on higher complexity methods. A striking example is the polyhedral model. Many advances in program restructuring have been achieved through this model which considers each instance of a statement as an integer point in a convenient space [16]. Most of the underlying methods, as data dependence analysis [8, 22], transformation computation [20, 11] or code generation [15, 24] exhibit worst-case exponential complexity.

It is not easy to conclude about the scalability of such techniques. The litterature is full of algorithms with high complexity which present a very good practical behaviour (the *simplex* algorithm is probably the most famous). Polyhedral code generation has an intrinsic worst-case complexity of $3^{n\rho}$ polyhedral operations (themselves associated with NP-complete problems), where n is the number of statements, and ρ the maximum loop depth. Nevertheless, input programs are not randomly generated. Most of the time, human-written codes show simple control, loop nests wih low depth and which enclose few statements. Such properties make it possible to regenerate, through the whole source-to-polyhedra-to-source framework, well known benchmark codes with hundreds of statements per static control compute kernel (in the SPECfp2000 benchmarks) in an acceptable amount of time [3].

Complex transformations may be automatically computed by a given optimizing compiler [5, 20, 4, 11] or discovered by a programmer with the help of an optimization environment [21, 6]. Their application diminishes the input program regularity and lead to a challenging code generation problem. The challenge may come either from the ability to compute any solution (because of a complexity explosion) or from the ability to find a satisfactory solution (because of a high resulting control overhead). To solve these problems *in practice*, a new experiment-driven study was necessary, starting from the best state-of-the-art code generation tool [3]. We analysed in depth a complex optimizing transformation sequence of the SPECfp2000 benchmark `Swim` that has been found by an optimization expert with the help of the URUK framework [6]. Our goal was to find properties of the transformations themselves that may be exploited to defer the complexity problem, and to improve the generated code quality.

To validate our approach, we studied and applied our methods to other complex problems that have been submitted by various teams from both industry and academia. Each of them uses its own strategy to compute transformations, which encourage the search for common transformation properties. `QR` has been provided by Reservoir Labs Inc. which develop the high level R-Stream compiler [13]. `Classen` has been submitted by the FMI laboratory of the University of Passau which develop the high level parallelization tool LooPo [18, 11]. `DreamupT3` has been supplied by the RNTL Project DREAM-UP between Thales Research, Thomson R&D and École des Mines de Paris [12]. General properties of these reference problems are shown in Figure 1. They proved to be quite different, spanning all typical sources of complexity in polyhedral code generation: each benchmark has its own reason to be challenging, e.g. high statement number for `Swim`, deep loop nests for `Classen`, big values that need multiprecision arithmetic to be to manipulated with `DreamupT3`.

	Reference problems			
Properties	`Swim`	`QR`	`Classen`	`DreamupT3`
Statement number	199	10	8	3
Maximum loop depth	5	3	8	2
Number of parameters	5	2	1	0
Scheduling dimensionality	11	7	7	1
Maximum coefficient value	60	5	4	1919

Fig. 1. General properties of reference problems

The paper is organized as follows. Section 2 introduces the polyhedral representation and transformation model, then presents the associated code generation problem. Section 3 positions our paper among related works. Section 4 investigates algorithmic scalability challenges and our solutions, driven by experimental evaluations of the four reference benchmarks. Section 5 addresses additional code generation challenges associated with code size reduction and efficiency; in particular, it presents the first modulo-condition elimination technique that succeeds for a large class of real-world schedules while avoiding code bloat due to multiversioning.

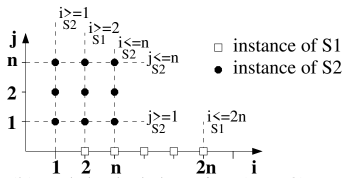
2 Overview of the Polyhedral Framework

This section presents both a quick overview of the polyhedral framework and notations we use throughout the paper. A more formal presentation of the model may be found in [23]. One usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code.

Our introductory example is a polynomial multiplication kernel. The syntactic form is shown in Figure 2(a). It only deals with control aspects of the program, and we refer to the two computational statements (array assignments) through their names, $S1$ and $S2$. To bypass the limitations of such representation (e.g. weak data dependence analysis, restriction to simple transformations), the polyhedral model is closer to the execution itself by considering *statement instances*. For each statement we consider the *iteration domain*, where every statement instance belongs. The domains are described using affine constraints that can be extracted from the program control. For example, the iteration domain of statement $S1$, called $\mathcal{D}_{S1}$, is the set of values (i) such that $2 \leq i \leq n$ as shown in Figure 2(b); a matrix representation is used to represent such constraints: $A \cdot x + A_p \cdot p \geq 0$, where A is the *iteration matrix*, x is the *iteration vector* (composed of the loop counters), A_p is the *parameter matrix* and p is the *parameter vector* (composed of the unknown constants and the scalar 1). In our example, $\mathcal{D}_{S1}$ is characterized by $\begin{bmatrix} 1 \\ -1 \end{bmatrix} \cdot (i) + \begin{bmatrix} 0 & -2 \\ 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix} \geq 0$.

```
for(i=2; i<=n; i++)
  z[i] = 0;                    // S1
for(i=1; i<=n; i++)
  for(j=1; j<=n; j++)
     z[i+j] += x[i] * y[j]; // S2
```

(a) Syntactic form



(b) Polyhedral domains ($n \geq 2$)

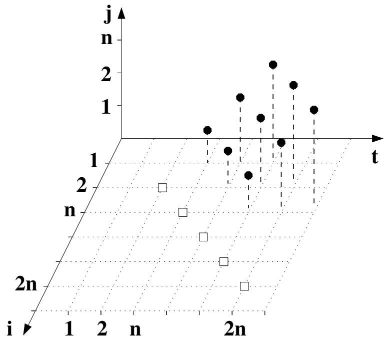
Fig. 2. A polynomial multiplication kernel and its polyhedral domains

In this framework, a transformation is a set of *affine scheduling functions* written $\theta(x) = T \cdot x + T_p \cdot p$. Each statement has its own scheduling function which maps each run-time statement instance to a logical execution date. In our polynomial multiplication example, an optimizer may notice a locality problem and discover a good data reuse potential over array z, then suggest $\theta_{S1}(i) = (i)$ and $\theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = (i + j + 1)$ to achieve better locality (see e.g., [4] for a method to compute such functions). The intuition behind such transformation is to execute consecutively the instances of $S2$ having the same $i + j$ value (thus accessing the same array element of z) and to ensure that the initialization of each element is executed by $S1$ just before the first instance of $S2$ referring this element. A transformation is applied in the polyhedral model by using the transformation formula shown in Figure 3(a) [3], where t is the *time-vector*, i.e. the vector of the scheduling dimensions. The resulting polyhedra for our example are shown in Figure 3(b) with the additional dimension t.

Once the transformation has been applied in the polyhedral model, one needs to generate the target code. The best syntax tree construction scheme consists in a recursive application of domain projections and separations [24, 3]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree. In our example, the first step is a projection onto the first dimension t, followed by a separation into disjoint polyhedra as shown on the top of Figure 4(a). This builds the first loop level of the target code (the loops with iterator t shown in Figure 4(b)). The

$$\left(\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right) \cdot \left(\frac{t}{x}\right) + \left(\frac{-T_p}{A_p}\right) \cdot p \; \begin{array}{c} = 0 \\ \hline \geq 0 \end{array}$$

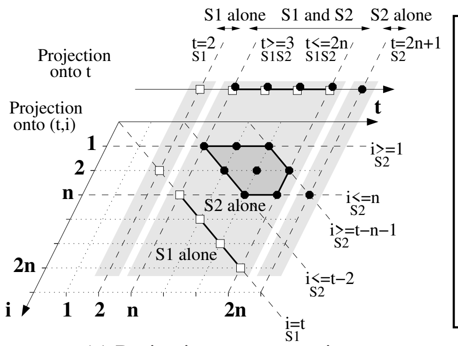
(a) Transformation formula



(b) Transformed polyhedra

Fig. 3. General transformation formula and its application

same process is applied onto the first two dimensions (on the bottom of Figure 4(a)) to build the second loop level and so on. The final code is shown in Figure 4(b) (the reader may care to verify that this solution does exploit at its best the temporal reuse of array z). Note that the separation step for two polyhedra needs three operations: $\mathcal{D}_{S1} - \mathcal{D}_{S2}$, $\mathcal{D}_{S2} - \mathcal{D}_{S1}$ and $\mathcal{D}_{S2} \cap \mathcal{D}_{S1}$, thus for n statements the worst-case complexity is 3^n.



(a) Projections an separations

```
// Indentation follows t,i,j domain dimensions
t=2; // A loop running once
  i=2;
    z[i] = 0;                          // S1
for(t=3; t<=2*n; t++)
  for(i=max(1,t-n-1); i<=min(t-2,n); i++)
    j = t-i-1;
      z[i+j] += x[i] * y[j]            // S2
  i=t;
    z[i] = 0;                          // S1
t=2*n+1;
  i=n;
    j=n;
      z[i+j] += x[i] * y[j];           // S2
```

(b) Target code

Fig. 4. Target code generation

3 Related Work

The history of code generation in the polyhedral model shows a constant growth in transformation complexity, from basic schedules for a single statement to general affine transformations for wide code regions. In their seminal work, Ancourt and Irigoin limited transformations to unimodular functions (the T matrix presented in Section 2 has determinant 1 or -1) and the code generation process was applicable for only one domain at once [1]. Several works succeeded in relaxing the unimodularity constraint to invertibility (the T matrix has to be invertible), enlarging the set of possible transformations [7, 19]. A further step has been achieved by Kelly et al. by considering more than one domain and multiple scheduling functions at the same time [15]. All these methods relied on the Fourier-Motzkin elimination method [26] to build the target code. Quilleré et al. showed how to use polyhedral operations based on the Chernikova Algorithm [17] instead, to benefit from its practical efficiency to handle bigger problems [24]. Recently, a new transformation policy has been proposed to allow general non-invertible, non-uniform, non-integral affine transformations [3]. Such freedom allowed to apply polyhedral techniques to much larger programs with very sophisticated transformations, and led to novel complexity, scalability and code quality challenges we discuss in this paper.

4 Code Generation Scalability

This section analyzes three important properties of affine schedules used in real-world program generation problems, then for each property, proposes an algorithmic solution to improve scalability.

4.1 Scalar Dimensions

There are many ways to specify a given transformation (or a given sequence of transformations) using affine schedules. Basically we can divide them in two families. The first kind, mono-dimensional schedules, describe the execution order thanks to functions with only one dimension. The second kind, multi-dimensional schedules, use several dimensions to express the ordering. Most of the time, the original domains are parametric, i.e., are bounded by (statically) unknown constants. For the first kind, this variety amounts to manipulating non-affine expressions. This is not the case with multi-dimensional schedules, when using at least as many dimensions as the original domain [9]. Moreover, using additional dimensions to explicitly order different statements onto a given dimension makes transformation manipulation easier [14, 6]. As a result, multi-dimensional schedules with more dimensions than original domains are quite often used to specify transformations. Figure 5 shows an example of a loop interchange transformation applied to the example in Figure 2(a) that may be achieved thanks to different schedules. $\rho(S)$ is the depth of the original statement, i.e., the number of dimensions of its original iteration domain.

Scheduling policy	θ_{S1}	θ_{S2}
Mono-dimensional	(i)	$(n+j*n+i)$
$\rho(S)$-dimensional	(i)	$(n+j,i)^T$
$(2*\rho(S)+1)$-dimensional	$(0,i,0)^T$	$(1,j,0,i,0)^T$

(a) Possible schedules for loop interchange

```
for(i=2; i<=n; i++)
  z[i] = 0;                /* S1 */
for(j=1; j<=n; j++)
  for(i=1; i<=n; i++)
     z[i+j] += x[i]*y[j]; /* S2 */
```

(b) Target code

Fig. 5. Loop interchange for polynomial multiplication using different schedules

Unified tranformation frameworks like UTF [14] or URUK [6] are good example of multi-dimensional schedule policies. Both ask for $(2\rho(S)+1)$ dimensions which allow them to be much more flexible. Nevertheless, using additional dimensions has a cost. In time: we will see that each dimension needs costly polyhedral operations (projection/separation/sorting). In space: each dimension implies (1) a new column in the constraint matrix, (2) as many rows as new constraints and (3) a new level in the generated code tree.

Most of the time, additional dimensions are *scalar*, i.e. they are constant for every scheduling functions. Because polyhedral operations on such dimensions are trivial, we systematically search and *remove them from the constraint matrix*, storing the scalar values in ad-hoc vectors. In the following, scalar dimensions will be implicitely stripped away from the schedule matrices. Polyhedral operations can proceed like in the classical algorithm, with the additional provision that they query ad-hoc scalar vectors prior to operating on polyhedra. In practice, before each separation step, we order the polyhedra according to the appropriate scalar vector components. Further steps of the code generation algorithm are applied onto lists of polyhedra having the same values for these components.

This optimization benefits from schedule properties without impacting expressiveness. It may dramatically reduce the number of polyhedral operations, improving both time and space complexity. Moreover, it also reduces the cost (in time and space) of every single polyhedral operation, by reducing matrix size. In practice, the actual benefits depend on the transformation policy: the more the constant scalar dimensions, the better the results. Also, this step has a very low complexity and thus does not degrade computation time even in worst case scenarios. Figure 6 shows the results when applying this optimization to our reference code generation problems. The scalar ratio gives the number of scalar dimensions with respect to the total number of dimensions, showing that the different teams which provided their problems do use scalar dimensions. This results into significant time and space improvement, except for the last program.

		Time			Space		
Benchmark	Scalar ratio	Original(s)	Scalar(s)	Speedup	Original(KB)	Scalar(KB)	Reduction
Swim	6/11	41.20	10.33	3.99×	17480	8128	2.15×
QR	4/7	19.47	2.44	7.98×	3012	988	3.05×
Classen	3/7	1.12	0.69	1.62×	1092	672	1.62×
DreamupT3	0/1	0.49	0.49	1.00×	160	160	1.00×

Fig. 6. Experimental results for scalar dimension removal

4.2 Node Fusion

When specifying transformations for a program with many statements, often is the case the processing is similar for several statements, at least for some dimensions. For instance, applying a given transformation (same schedules) to some statements of a given loop nest (same domains) allow to consider only one statement block. The modified version of the Quilleré algorithm [3] is given in Figure 7 and exploits the similarities of the transformations on certain dimensions for different statements.

Steps 4 and 5 create worklists that fully take advantage of the detection of scalar dimensions described in Section 4.1. Step 6 examines nodes of each job of the worklist and tries to fuse them into subworklists to reduce the number of elements given to the Quilleré algorithm as much as possible. Node fusion occurs at current depth on the projected domains and is guaranteed to exploit similarities between schedules at each nesting level independently. The complexity gain of Steps 4, 5 and 6 is difficult to quantify as it depends on the shape of the generated code itself and transformation similarities across different statements.

Considering a simple case with n statements in a loop nest level that can be blocked into c chunks of s_c statements with same scalar components. Suppose each chunk can further be blocked into b_c blocks of $l_{b_c} \leq s_c$ statements with same projected domain. This translates to $\sum_{b_c} (\text{Quilleré}(l_{b_c}))$ instead of $\text{Quilleré}(n)$ which stands for a call to the Quilleré separation algorithm that has a worst-case complexity of 3^n. Furthermore,

```
CodeGeneration: builds an AST (Abstract Syntax Tree) scanning a list of polyhedra
Input:
  node: flat AST holding the domains to scan
  context: static context (known constraints met by the parameters)
  depth: the nesting level
Output: An AST scanning the polyhedra in the lexicographic order

  nodelist ← ∅; worklist ← ∅; subworklist ← ∅
  while node has successors
1   Intersect node.domain with the context
2   Project intersected domain on the depth outermost dimensions and on parameters
3   node ← node.next

  if nodes have scalar values at depth and they are different
4   Sort nodes according to their scalar values at depth
5 worklist ← partition nodes by scalar values

  foreach job in worklist
6   fusedlist ← Fuse nodes of job with the same projected intersected domain
7   separatedlist ← Apply Quilleré's separation step to fusedlist
8   sortedlist ← Sort separatedlist according to the lexicographic order
    foreach ASTnode in sortedlist
      if ASTnode.domain dimensionality > depth
9       ASTnode.inner = CodeGeneration(innerlist, context, depth+1)
10    Enqueue ASTnode to AST

  return AST
```

Fig. 7. Code generation algorithm

Step 8 also benefits from the reduction above and allows for $\sum_{b_c}(\mathrm{Sort}(l_{b_c}))$ instead of $\mathrm{Sort}(n)$ which stands for a call to a function sorting n polyhedra that also has an exponential worst case complexity. Experimental results are summarized in Figure 8. As expected, this technique is quite useful for large problems like `Swim`.

	Time			Space		
Benchmark	Original(s)	Fused(s)	Speedup	Original(KB)	Fused(KB)	Reduction
Swim	41.20	5.90	6.98×	17480	5048	3.46×
QR	19.47	19.17	1.02×	3012	2992	1.01×
Classen	1.12	1.03	1.09×	1092	1060	1.03×
DreamupT3	0.49	0.49	1.00×	160	160	1.00×

Fig. 8. Experimental results on node fusion

4.3 Domain Iterators

It is well known that code generation is easier when restricting the problem to invertible schedules [28, 24]. CLooG was the first tool to seamlessly manage non-invertible schedules, at the cost of additional recursion steps, polyhedral projections and larger matrix sizes in Quilleré's algorithm [2, 3]. For scalability reasons, we propose to detect invertibility conditions and refine the recursive AST traversal automatically. Indeed, when considering invertible transformations, the value of the original domain iterators (used, e.g., in the statement bodies) according to the target space iterators can be efficiently obtained via matrix inversion (instead of recursive polyhedral projections).

Let $\theta(x) = T \cdot x + T_p \cdot p$ be a schedule transformation where T is invertible, and consider an iteration domain $\mathcal{D} : A \cdot x + A_p \cdot p \geq 0$. The transformed domain $\mathcal{T}$ (see Figure 3(a)) can be broken down into two distinct components:

– a polyhedron to scan (Figure 9) obtained by projecting $\mathcal{T}$ on time iterators and parameters only;
– an *inverted scatter matrix* (ISM) that associates, locally to each statement, the expression of the domain iterators as invertible functions of time iterators and parameters. When T is non-unimodular, T^{-1} has rational coefficients. Let $(d_{i,j})$ be the denominators of T^{-1}, by taking $\lambda_i = \mathrm{lcm}(d_{i,\bullet})$ we define $\Lambda = \mathrm{Diag}(\lambda_i)$ as the diagonal matrix where the diagonal element of the i^{th} line is λ_i. The left multiplication by $(\frac{\Lambda T^{-1}}{I})$ of the expression of $\mathcal{T}$ (Figure 3(a)) yields the final integer form of ISM shown in Figure 10.

$$\mathcal{T} \perp \left(\begin{array}{c|c|c} t & 0 & 0 \\ \hline 0 & 0 & p \end{array}\right)$$

Fig. 9. Simplified time-extended domain

$$\left(\Lambda T^{-1} \,\middle|\, -\Lambda\right)\left(\frac{t}{x}\right) - \Lambda T^{-1} T_p p = 0$$

Fig. 10. ISM to recover the domain iterators

The benefits brought to the separation algorithm are threefold and contribute to possibly exponential complexity gains:

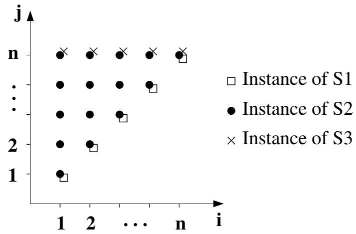
– it is straightforward to write domain iterators as expressions of time iterators and parameters from Figure 10 instead of performing costly polyhedral projections on each domain iterator;
– the column number of each polyhedron to scan is reduced by the number of domain iterators (potentially half the original size if there are no parameters);
– the height of the generated AST is reduced on each path to every statement by the same amount above. However the paths subject to reduction are linear and save no branches from the original AST but still save polyhedral projections.

The `Swim` benchmark has invertible schedules only; this is a strong assumption of the URUK framework [6]. Yet this was not enforced by the scheduling strategies of the 3 other benchmarks. Interestingly, it seems possible to extend this domain iterator elimination technique to all kinds of non-invertible schedules, combining Gaussian elimination steps with polyhedral projections. Yet this is left for future work. In the current state of our implementation, this optimization could only be applied to `Swim`, yielding 36% speedup for code generation time and 57% reduction in memory usage.

4.4 If Conditional Hoisting

Under complex transformation sequences, the top-down part of Quilleré's code generation algorithm [24] yields `if` conditionals that greatly hamper the quality of the generated code and thus, its execution time. Figure 11 exhibits this behavior on a basic example: generating a code for scanning the polyhedra of Figure 11(a) using the algorithm in Figure 7 would lead to the code in Figure 11(b). This figure shows internal guards leading to a high control-overhead.

The approach presented in [24] for removing inner `if` conditionals and generating the better code in Figure 11(c) consists of a backtracking call to the separation procedure. Although it proved successful at performing its primary task, its side effects can yield unnecessary computation and code bloating. The forementionned algorithm lacks the capability of factorizing similar conditionals. Examine a node at depth d after the separation phase. Assume the separation has generated an inner conditional c which




```
for(i=1;i<=n;i++)
  if(i == n)
    S1(j=n)
    S2(j=n)
    S3(j=n)
  if(i <= n-1)
    S1(j=i)
    S2(j=i)
  for(j=i+1;j<=n-1;j++)
    S2
  if(i <= n-1)
    S2(j=n)
    S3(j=n)
```

```
for(i=1;i<=n-1;i++)
  S1(j=i)
  S2(j=i)
  for(j=i+1;j<=n-1;j++)
    S2
  S2(j=n)
  S3(j=n)
S1(i=n,j=n)
S2(i=n,j=n)
S3(i=n,j=n)
```

(a) Domains to scan (b) Before if-hoisting (c) After if-hoisting

Fig. 11. Removing internal guards with if-hoisting

depends only on the i, $i < d$, first dimension iterators. During the backtracking called on depth d, the algorithm in [24, 3] performs separation regardless of the condition c. Therefore, costly polyhedral operations have been made while only a separation at depth i was necessary. Focusing only on conditionals also avoid to achieve versionning for triangular loops which do not execute for only few values of the outer loop counters. For instance, we can see that in Figure 11(c) the j-loop do not iterate for $i = n - 1$, removing this negligible control overhead whould have led to a 50% code growth.

Our solution boils down to a depth-first traversal of the AST, fetching all the conditionals of subsequent domains for the current nesting level, *factorizing* them by performing polyhedral separation (intersection and difference) *on conditionals relevant to the current depth only*, and intersecting these newfound conditionals with the current domain, duplicating the underlying AST structure. The algorithm, which intervenes as a post pass after separation guarantees no unnecessary cuts are performed and therefore avoids unnecessary code explosion. Figure 12 shows the duplication factor results on the four reference benchmarks, i.e., the number of computational statements in the generated code divided by the number of statements in the polyhedral representation, a reasonable metric for code quality [2]. These results show strong code size reductions can be achieved through our improved `if`-hoisting phase. The relatively low duplication factor for `Swim` (2.5) is also a very good indication of the applicability and scalability of polyhedral techniques to larger optimization and parallelization problems. Eventually, to better isolate the effect of this optimization, the last row (Figure 12) reports results for the simple one-statement matrix multiplication, applying three-dimensional tiling and shifting through the URUK framework [6]. It incurs major (yet unavoidable) code bloat, but our technique reduces it by a factor of 2.5.

Benchmark	Original dup. factor	`if`-hoisting dup. factor	Reduction
Swim	2.5	2.5	1
QR	107	35	3
Classen	11.5	9.6	1.2
DreamupT3	23.3	4	5.8
MxM	175	69	2.5

Fig. 12. Experimental results with `if`-hoisting

5 Code Quality

Beyond code generation performance, addressing real-world problems raises generated code quality issues that may not directly emerge from smaller, academic examples. This section investigates two of them: extending code generation to implement a smarter loop unrolling strategy, and building on this extension to achieve a major step in code generation for strided domains and reindexed schedules.

5.1 Enabling Strip-Mining for Unrolling

In most cases, loop unrolling can be implemented as a combination of *strip-mining* and *full unrolling* [27]. Strip-mining itself may be implemented in several ways in a polyhedral setting. Following our earlier work in [6] and calling b the strip-mining factor, we choose to model a strip-mined loop by dividing the iteration span of the outer loop by b instead of leaving the bounds unchanged and inserting a non-unit stride b:

```
for(i=ℓ(x); i<=u(x); i++)
```

$\xrightarrow{\text{strip-mine}\,(b)}$

```
for(t1=⌈ℓ(x)/b⌉; t1<=⌊u(x)/b⌋; t1++)
 for(t2=max(ℓ(x),b*t1); t2<=min(u(x),b*t1+b-1); t2++)
```

This design preserves the convexity of the polyhedra representing the transformed code, alleviating the need for specific stride-recognition mechanisms (based, e.g., on the Hermite normal form).

In Figure 13(b) we can see how strip-mining by a factor of 2 the original code of Figure 13(a) yields an internal loop with non-trivial bounds. It can be very useful to unroll the innermost loop to exhibit register reuse (a.k.a. register tiling), relax scheduling constraints and diminish the impact of control on useful code. However, unrolling requires to cut the domains so that `min` and `max` constraints disappear from loop bounds. Our method is adapted the one presented for hoisting `if` conditionals; the difference lies in the selection of conditionals. For the purpose of if-hoisting (see Section 4.4), we just had to pick the constraints that did not concern the node at current depth. Here we focus on finding conditionals (lower bound and upper bound) for the current depth, *such that their difference is a non-parametric constant*: the unrolling factor. Hoisting these conditionals actually amounts to splitting the outer strip-mined loop into a kernel part where the inner strip-mined loop will be fully unrolled, and a remainder part (not unrollable) spanning at most as many iterations as the strip-mining factor. In our example, the conditions associated with a constant trip-count (equal to 2) are `t2>=2*t1` and `t2<=2*t1+1` and are associated with the kernel, separated from the prologue where `2*t1<M` and from the epilogue where `2*t1+1>N`. This separation leads to the more desirable form of Figure 13(c).

Finally, instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a lazy way, avoiding the added (exponential) complexity on the separation algorithm. This approach relies on a preliminary strip-mine step that determines the amount of partial unrolling.

5.2 Removing Modulo Conditions

When the transformed domains $\mathcal{T}$ (see Figure 3(a)) are $\mathbb{Z}$-polyhedra (a.k.a. lattice polyhedra), the generated code shows modulo conditions. The modulo guards guarantee that only the iterations that belong to the original domain are scanned in the generated code.

```
for(t1=M; t1<=N; t1++)
  S1(i = t1);
```
(a) Original code

```
for(t1=M/2; t1<=(N+1)/2; t1++)
  for(t2=max(M,2*t1);
      t2<=min(N,2*t1+1); t2++)
    S1(i = t2);
```
(b) Strip-mining of 2

```
if(M%2==1)
  S1(i = M);
for(t1=(M+1)/2; t1<=(N-1)/2; t1++)
  S1(i = 2*t1);
  S1(i = 2*t1+1);
if(N%2==0)
  S1(i = N);
```
(c) Separation & unrolling

Fig. 13. Strip-mining and unrolling transformation

For instance, if the ISM of a statement S (see section 4.3) that gives the value of the original domain iterators (e.g., i) according to the transformed space iterators (e.g., t) gives $2i = t$, the execution of the statement S will be guarded with `if` (t%2 == 0). This situation happens either when the transformation matrices T are not unimodular or when the original domains $\mathcal{D}$ are $\mathbb{Z}$-polyhedra, e.g., in some kinds of strip-mined loops[1]. Both cases boil down to the same code generation problem. For space reasons, we will only detail our solution in the case of invertible, non-unimodular schedules.

The consequence of generating modulo guards is to introduce a high control overhead. Many works focused on finding solutions to avoid them. The first idea was to compute the convenient loop strides. At first it was done using the Hermite Normal Form [19, 28, 7, 25], but this was limited to only one domain, then by considering the transformation expression itself [15, 2], but some guards cannot be removed in this way. More recent methods suggest to use strip-mining for one domain [10], or to find equivalent transformations with convenient additional dimensions when this is possible [11], or to unroll the loops according to a convenient unroll factor in the case where modulo guards depend on only one loop counter [11]. Here we give a general algorithm to drastically reduce the number of modulo guards inside the loops and even void them all in the *loop kernels*.

Consider a simple example with two statements, where S1 has the one-dimensional schedule $2t_1 - 5$ and S2 has the one-dimensional schedule $3t_1$. In other words, the rate of S1 is 50% higher than S2 and is shifted ahead by 5 iterations. This example is derived from the low-level scheduling and code generation for a software-pipelined FIR filter, where one functional unit (a multiplier in S1) is needed at a 50% higher rate than a another one (an adder in S2), and S2 depends on S1. Due to the combined reindexing (factors 2 and 3 in the schedule) and shifting (by 5 iterations), traditional techniques to avoid modulo expressions cannot be applied [2], and existing code generators yield the inefficient code of Figure 14. Our technique eliminates modulo expressions completely from the kernel part (the hot path) of the generated code, without code bloat, and generates the much more efficient version in Figure 15. On this simple example, our technique achieves a 67% speedup in generated code execution-time, with respect to the more naive one with modulo expressions.

In the general case, the main problem resides in the lower bound of the scattered domain [7, 25, 28] whose value *modulo the stride factor* must be known in order to exhibit a regular pattern in the loop body. This lower bound can be viewed as a *pattern alignment synchronization barrier* for S1 and S2. Indeed, parametric schedules with

[1] Although it is always possible to express strip-mining with convex polyhedra only, see Section 5.1.

```
(...)
// software pipeline kernel
for (t1=5; t1<=2*N-2; t1++)
  if ((t1-5)%3 == 0)
    S2(i = (t1-5)/3);
  if (t1%2 == 0)
    S1(i = t1/2);
(...)
```

Fig. 14. Traditional code generation

```
// prologue
S2(0);
// kernel code with S1 and S2 synchronized modulo 6
for (t1=1; t1<=floord(N-4,3); t1++)
  S1(i = 3*t1); S2(i = 2*t1-1); // t2%6=0
  S1(i = 3*t1+1);               // t2%6=2
  S1(i = 3*t1+2);               // t2%6=3
  S2(i = 2*t1);                 // t2%6=4
// epilogue
for (t1=ceild(N-3,3); t1<=floord(N-1,3); t1++)
  for (t2=6*t1; t2<=2*N-2; t2++)
    if ((-t2+5)%3 == 0)
      S2(i = (t2-5)/3);
    if (-t2%2 == 0)
      S1(i = t2/2);
```

Fig. 15. Our solution for the software-pipelined kernel

non-unit stride factors may generate as many different loop body patterns as the least common multiplier of these strides; notice these patterns are *not* identical (in general) up to loop body "rotations". The only solution to thoroughly eliminate modulo conditions is multi-versioning, but it results in severe code bloat for stride factors over 2 or 3.

Our approach consists in forcing *pattern synchronization* by strip-mining the original loop by a factor that is yet to determine. This amounts to extracting a prologue and an epilogue from the unrollable kernel, yielding the much more efficient solution of Figure 15. Using this method, the prologue and epilogue still contain internal modulo conditions whereas the kernel (where the vast majority of the execution time is spent) can be unrolled. This approach is effective on a large class of "well-behaved" schedules. We will argue at the end of this section that the other "ill-behaved" schedules are intrinsicly code-bloating if modulo expression elimination is to be attempted.

The previous case having the sole purpose of stating the problem simply, we now outline the general algorithm. This step takes place after the separation, `if`-hoisting, and lazy unrolling steps. From the *Inverse Scatter Matrix* (ISM) shown in Figure 10, we can derive that the i^{th} original loop iterator x_i corresponding to a given statement S can be expressed thanks to the i^{th} line of its ISM formula: $\lambda_i \cdot x_i = \left(\sum_j (k_{i,j} \cdot t_j) + C\right)$, where C is the constant parametric part. It follows, a modulo condition that rules the execution of S is $\left(\sum_j (k_{i,j} \cdot t_j) + C\right) \bmod \lambda_i = 0$. Let us first assume that C is known at compile time. The point is to statically determine the values of $(k_{i,j} \cdot t_j) \bmod \lambda_i$ for all i and j to be able to remove all the modulo guards. For that purpose, for each node of the AST at depth j, the time dimension t_j will be unrolled by the least common multiplier over all statements under this node (at depth j) of

$$\mathrm{lcm}_j = \operatorname*{lcm}_{\{i \mid k_{i,j} \neq 0\}} \left(\lambda_i / \gcd(k_{i,j}, \lambda_i)\right).$$

Unrolling by this factor yield as many instances of t_j for which we statically know the value modulo λ_i. For a given loop node at depth d, the least common multiplier of all such unrolling factors yields the global unrolling factor lcm_j that guarantees static elimination of all internal modulo conditions. To enable unrolling, a new time dimension is introduced by strip-mining by lcm_j. This new dimension scans the same points as the old time dimension, with the additional property that its first iteration is divisible by lcm_j, thus achieving the required *synchronization of all statements to a statically known pattern*. Building on the strip-mining method introduced in Section 5.1, the strip-mined loop is actually split into a prologue, a so-called *zero-aligned kernel*, and an epilogue. By construction, the zero-aligned kernel has the important property that *its outer strip-mined loop scans multiples of lcm_j only*. Thanks to this property, and having fully unrolled the inner strip-mined loop, we may *statically evaluate* the remainder of the division of the *inner strip-mined loop's iterator* by lcm_j. Applying this systematically to all depths where lcm_j is greater than 1 allows all modulo conditions to be removed *from the zero-aligned kernel only*.

```
RemoveModuloGuards: removes modulo conditionals from loop kernels
Input:
  node: AST root node
  depth: the depth of the modulo conditional
Output: an AST without modulo conditionals in loop nest kernel

  nodelist ← empty list
  while node has successors
    if node is a for loop
1     copute lcm_depth
      if lcm_depth > 1
2       kernel.inner ← new time dimension between t_depth and t_depth+1 with constraints
          lcm_depth × t_depth ≤ t_new ≤ lcm_depth × t_depth + (lcm_depth − 1)
3       Update all the statement informations (domains and ISMs) with the new dimension
4       Strip-mine and partition node.domain in prologue, zero-aligned kernel, and epilogue
5       Enqueue prologue, kernel and epilogue to nodelist
6       Unroll kernel with respect to t_new
7       RemoveModuloGuards(kernel.inner.inner, depth+2)
      else
8       RemoveModuloGuards(node.inner, depth+1)
    else node is a statement
9       Prune node off the AST if needed
    node ← node.next

  return nodelist
```

Fig. 16. `RemoveModuloGuards` Algorithm

The algorithm in Figure 16 describes how to introduce new time dimensions and unroll them so as to eliminate modulo conditions. Step 9 is actually not trivial. When reaching the leaves of the AST, we need to determine which modulo guards have been simplified, which ones are still necessary and which ones have become unfeasible. Having strip-mined (and unrolled) by the factor $\mathrm{lcm}_{\mathrm{depth}}$, we have forced newly created time iterators on the path to the innermost kernel to be divisible by λ_i. If all the components of an ISM line i are divisible by λ_i, then the modulo condition is always true and needs not to be printed. If all the components are divisible by λ_i but not the constant part, the modulo condition is always false and the statement should be pruned. In the last case, the modulo condition for line i needs to be printed, but at least its expression simpler (and faster to evaluate) than it would have been without strip-mining and unrolling.

Had we wished to fully unroll and had we used versioning, we could have generated an unreasonable number of versions (up to the factorial of $\mathrm{lcm}_{\mathrm{depth}}$). Our algorithm manages to fully unroll the kernel only, where most computation time is spent, while the prologues and epilogues (with modulo conditions) hold at most $\mathrm{lcm}_{\mathrm{depth}} - 1$ iterations.

When the value of *constant parametric shift component C* modulo $\mathrm{lcm}_{\mathrm{depth}}$ is not statically known, it is impossible to statically determine an interleaving pattern. Synchronizing the values of time iterators modulo $\mathrm{lcm}_{\mathrm{depth}}$ does not help and even leads to the insertion of internal modulo conditions. Nonetheless, one can argue on the interest of schedules that do not exhibit a regular pattern: the interleaving of statements itself totally changes with the values of parameters, hence is intrinsincly tied to multi-versioning.

6 Putting it All Together

Let us combine all the previous optimizations and summarize the total improvements in code generation time, memory usage and generated code size. To further stress the scalability of our tool, we added a more complex optimization of the `Swim` benchmark, called `Swim+`, in its most general setting with 5 parameters (without context).

	Time			Space			Code size		
Benchmark	Orig.(s)	Opt.(s)	Speedup	Orig.(KB)	Opt.(KB)	Reduction	Orig.(Lines)	Opt.(Lines)	Reduction
Swim	41.20	2.41	17.09×	17480	2380	7.34×	830	764	1.09×
Swim+	1219.67	21.62	56.41×	322624	22180	14.55×	17791	12041	1.48×
QR	19.47	2.42	8.05×	3012	988	3.05×	4733	1432	3.33×
Classen	1.12	0.25	4.48×	1092	272	4.01×	130	105	1.24×
DreamupT3	0.49	0.20	2.45×	160	160	1.00×	382	68	5.62×

Fig. 17. Summary of experimental results

7 Conclusion

The polyhedral model is a powerful framework to reason about high level loop transformations. Recently, new algorithms made it possible to compute the target code for hundreds of statements while this code generation step was expected not to be scalable. Unfortunately, these improvements allowed the exploration of larger, more complex optimization and parallelization problems, which in turn raised several scalability and code quality challenges.

We presented scalable code generation methods that make possible the application of complex program transformations to real-world computation kernels with up to 199 statements. By studying the transformations themselves, we show how it is possible to benefit from their properties to dramatically improve both code generation quality and space/time complexity. Moreover, building on these algorithmic improvements, we proposed a new algorithm to generate more efficient (conditional-free) code for strided domains and reindexed schedules.

We believe these improvements — implemented in the latest versions of the CLooG [3] and WRaP-IT/URUK [6] frameworks — will initiate an other vertuous cycle towards allowing polyhedral techniques to bring dramatic improvements in the effectiveness of optimizing and parallelizing compilers.

References

1. C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.
2. C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE Intl. Symp. on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.
3. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE Intl. Conf. on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.
4. C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 Intl. Conf. on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.

5. P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3):421–444, 1998.
6. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM ICS'05 International Conference on Supercomputing*, pages 151–160, Cambridge, june 2005.
7. A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.
8. P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.
9. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Int. Journal of Parallel Programming*, 21(6):389–420, december 1992.
10. B. Franke and M. O'Boyle. A complete compiler approach to auto-parallelizing c programs for Multi-DSP systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(3):234–245, march 2005.
11. M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.
12. I. Hurbain, C. Ancourt, F. Irigoin, M. Barreteau, J. Mattioli, and F. Paquier. A case study of design space exploration for embedded multimedia applications in SoCs. Technical Report A-361, CRI – École des Mines de Paris, february 2005.
13. U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. Ho Ahn, P. Mattson, and J. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, august 2003.
14. W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, 1993.
15. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.
16. D. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.
17. H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.
18. C. Lengauer. Loop parallelization in the polytope model. In *Int. Conf. on Concurrency Theory, LNCS 715*, pages 398–416, Hildesheim, August 1993.
19. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.
20. A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *PoPL'24 ACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, January 1997.
21. R. Müller-Pfefferkorn, W. Nagel, and B. Trenkler. Optimizing cache access: A tool for source-to-source transformations and real-life compiler tests. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 72–81, Pisa, august 2004.
22. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, august 1991.
23. W. Pugh. Uniform techniques for loop optimization. In *ICS'5 ACM International Conference on Supercomputing*, pages 341–352, Cologne, june 1991.
24. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.
25. J. Ramanujam. Beyond unimodular transformations. *J. of Supercomputing*, 9(4):365–389, 1995.
26. A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., 1986.
27. M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1995.
28. J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.