Polyhedral AST generation is more than scanning polyhedra

GROSSER TOBIAS, INRIA and ÉCOLE NORMALE SUPÉRIEURE [1]
VERDOOLAEGE SVEN, INRIA, ÉCOLE NORMALE SUPÉRIEURE and KU Leuven[2]
COHEN ALBERT, INRIA and ÉCOLE NORMALE SUPÉRIEURE

Abstract mathematical representations such as integer polyhedra have shown to be useful to precisely analyze computational kernels and to express complex loop transformations. Such transformations rely on Abstract Syntax Tree (AST) generators to convert the mathematical representation back to an imperative program. Such generic AST generators avoid the need to resort to transformation-specific code generators, which may be very costly or technically difficult to develop as transformations become more complex. Existing AST generators have proven their effectiveness, but they hit limitations in more complex scenarios. Specifically, (1) they do not support or may fail to generate control flow for complex transformations using piecewise schedules or mappings involving modulo arithmetic; (2) they offer limited support for the specialization of the generated code exposing compact, straightline, vectorizable kernels with high arithmetic intensity necessary to exploit the peak performance of modern hardware; (3) they offer no support for memory layout transformations; (4) they provide insufficient control over the AST generation strategy, preventing their application to complex domain-specific optimizations.

We present a new AST generation approach that extends classical polyhedral scanning to the full generality of Presburger arithmetic, including existentially quantified variables and piecewise schedules, and introduce new optimizations for the detection of components and shifted strides. Not limiting ourselves to control flow generation, we expose functionality to generate AST expressions from arbitrary piecewise quasi-affine expressions which enables the use of our AST generator for data-layout transformations. We complement this with support for specialization by polyhedral unrolling, user-directed versioning, and specialization of AST expressions according to the location they are generated at, and complete this work with fine-grained user control over the AST generation strategies used. Using this generalized idea of AST generation, we present how to implement complex domain-specific transformations without the need to write specialized code generators, but instead relying on a generic AST generator parametrized to a specific problem domain.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processor – Compilers, Optimizations

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Polyhedral compilation, code generation, unrolling, index set splitting, Presburger relations

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (2015), 50 pages.
DOI:http://dx.doi.org/10.1145/2743016

1. INTRODUCTION

The development of high-level optimizations for domain-specific or general purpose compilers can often be conceptually divided into two parts: the design of a high-level

[1]Part of this work was performed while working for ETH Zurich.
[2]Part of this work was performed while working for LIACS, Leiden.

optimization strategy and the generation of program code according to this optimization strategy. In many cases, the interesting scientific contribution is the new optimization strategy. However, in practice, significant efforts are put into the generation of efficient program code.

For loop programs with affine (and even non-affine [Venkat et al. 2014]) control flow, it is common to automatically generate optimized program code from an abstract description that models each statement instance (i.e., each dynamic execution of a statement inside a loop nest) and each array element individually through the use of a compact representation such as polyhedra [Loechner and Wilde 1997] or Presburger relations [Pugh and Wonnacott 1994]. Optimizations, for example those of Pugh and Rosser [1997; Wonnacott [2002; Bondhugula et al. [2008; Kong et al. [2013; Zuo et al. [2013] and Bandishti et al. [2012], are described by modifying an abstract schedule that defines the execution order of the individual statement instances in a program. According to this schedule, imperative program code is (re)generated using a technique called polyhedral scanning [Ancourt and Irigoin 1991], code generation [Kelly et al. 1995; Bastoul 2004], or, more accurately, Abstract Syntax Tree (AST) generation. Decoupling the optimization and program generation steps not only reduces the time needed to implement a certain optimization strategy, it also speeds up the evaluation of new optimization strategies [Pouchet et al. 2007; Pouchet et al. 2008]. Furthermore, being able to describe transformations on a highly abstract level enables the development of complex transformations [Grosser et al. 2014] relying on the AST generator to generate efficient imperative code.

Even though AST generators have many benefits, existing approaches focus on control flow generation [Bastoul 2004; Kelly et al. 1995; Chen 2012], provide only rudimentary support for the specialization of the generated expressions, and limited control over code size vs. control overhead. These limitations often prevent their wider usage. Missing support for generating user-provided AST expressions, e.g., to describe memory locations, prevents their application to data-layout transformations [Wonnacott 2002; Yuki et al. 2012; Henretty et al. 2013] or, to the mapping of data to software managed caches [Holewinski et al. 2012]. Also, existing AST generators may produce multiple code versions according to specific parameter values, to reduce control overhead, but existing approaches do not natively support the generation of specialized code. This would be particularly helpful for the separation of full and partial tiles [Ancourt and Irigoin 1991; Goumas et al. 2003; Kim et al. 2007] or for the generation of specialized code to handle possibly different conditions at iteration space boundaries. Instead, versioning has to be enforced by generating several distinct copies of each statement in the input description [Chen et al. 2008]. Similarly, performing unrolling during AST generation is only possible by duplicating statements in the input description [Chen et al. 2008; Bondhugula et al. 2008; Shirako et al. 2014]. Besides being conceptually unsatisfying, duplicating statements causes serious problems. Firstly, by purposefully hiding the fact that statements are identical, the AST generator is forced to generate duplicate code for them in all cases, missing redundancies in complex expressions and missing opportunities to factor colder parts of the code. Secondly, duplicating statements increases the complexity of the polyhedral operations involved in the generation of imperative control flow, supporting optimizations such as full/partial tile separation, and supporting expression specialization or simplification for modulo arithmetic. If we now wish to minimize code size for colder parts of the iteration space (e.g., the partial tiles), we run into the next limitation. Even though AST generators provide basic control over the desirable aggressiveness in separating statements or control flow specialization (conditional hoisting), the level of control is way too coarse-grained in existing methods and tools. Also, no guarantees are given about the maximal number of loop nests and the maximal number of statements generated, which is problematic for scenarios where code size is a major concern, such as AST generation for many-core targets with software-managed caches, embedded processors, and high-level synthesis [Zuo et al. 2013]. Overall, existing approaches and tools are in many ways not yet mature for complex AST generation problems.

This work presents an integrated AST generation approach that, in addition to classical control flow generation, allows the generation of AST expressions from arbitrary user-provided piecewise affine expressions.[3] We define a fine-grained "option" mechanism that enables the user to request maximal specialization where needed while retaining control over code size. To enable aggressive specialization, we allow the user to instruct the AST generator how to version the code, we provide an integrated polyhedral unrolling facility, and we make sure that AST expressions are specialized according to the context they are generated in. Doing so is essential to correctly model the floor-division and modulo arithmetic arising from abstract transformations of the program, and to cast these expressions to efficient remainder and integer divisions, or to lower-complexity operations, as provided by existing instruction set architectures and programming languages. Finally, we present a miscellaneous set of optimizations that improve the quality of the generated code, in comparison to existing polyhedral scanning tools, but also optimizations made necessary to cover the wider application scenarios of our AST generator.

Our contributions are as follows:

— AST generation with complete support for *Presburger relations* including support for *piecewise schedules*, and their use in expressing *index set splitting as a schedule-only transformation*.
— An aggressive simplification of *AST expressions* generated from *piecewise quasi-affine expressions* within the context of their position in the AST, including the *detection of modulo and division operations*. The generation of simplified AST expressions is not only used to construct loop bounds and `if` conditionals from within the AST generator, but is also exposed to the user, who can use this functionality to generate custom index expressions and run-time checks.
— *Fine-grained options* to control AST generation including an *atomic* option that can be used to control code size and to ensure that no program statements are duplicated.
— *Specialization* through *polyhedral unrolling* and *user directed versioning*, in particular the user can specify a subset of the schedule space (e.g., full tiles) that should be isolated from the rest of the schedule space (i.e., partial tiles).
— Algorithms for *improved stride detection* and *the detection of reorderable components*.
— AST generation for structured schedules expressed as *schedule trees*.
— Evaluation in an advanced domain-specific optimizer and comparison to state-of-the-art code generation techniques.

The remaining content of this paper is organized as follows. Section 2 gives a high-level overview of our new AST generation approach and presents new, illustrative use cases. We then present theoretical background in Section 3, the data structures involved in Section 4, our core AST generation approach in Section 5, and its extension to schedule trees in Section 6. We finish with a set of experiments in Section 7, the discussion of related work in Section 8, and the conclusion in Section 9.

[3]The entire approach is available at http://repo.or.cz/w/isl.git in the development version isl-0.14-368-g23e8573, which will be included in the next releases of `isl`.

2. A NEW APPROACH TO AST GENERATION

To give an idea of the new AST generation concepts proposed in this work, we present them in the context of a complex AST generation scenario. One such ideal scenario is our recent work on hexagonal/parallelogram tiling [Grosser et al. 2014], a domain specific optimization for generating efficient CUDA code for iterative stencil computations. It is implemented on top of PPCG [Verdoolaege et al. 2013], a generic C to CUDA/OpenCL translator, which uses Presburger relations to describe both the computation itself and the program transformation to apply. Taking advantage of such an abstract mathematical description, a new tiling scheme has been developed that involves complex geometric shapes to address the most important performance issues of compiling iterative stencils for GPUs, including the usage of shared memory, the optimization of data-transfers, the increase of arithmetic intensity, the exploitation of multiple levels of parallelism, and the avoidance of thread divergence. In the following paragraphs, we show how we obtained highly efficient code using our new generic AST generation approach without the need to develop a domain- or optimization-specific generator.

When translating C code to CUDA, we start from code consisting of compute statements and loops. To simplify the exposition, let us first assume in this section that the program consists of a single perfectly nested set of loops, with one outer sequential loop, a set of inner parallel loops, and a single compute statement. To generate CUDA code for this computation it is necessary to obtain a set of kernels that can be launched sequentially and that each expose two levels of parallelism: coarse grained parallelism, which will be mapped to so-called CUDA thread blocks, and fine grained parallelism, which will be mapped to so-called CUDA threads. To obtain these two levels of parallelism we divide the set of individual computations (statement instances) enumerated by these loops into subsets (tiles). We do this by computing a polyhedral schedule that enumerates the set of statement instances with two groups of loops. A set of outer loops that enumerate the tiles (tile loops) and a set of inner loops (point loops) that enumerate the statement instances that belong to a certain tile. The first AST generation problem we encounter is that the hybrid-hexagonal schedule defining the tile shapes decomposes the computation into phases and applies to each phase a different schedule. This results in a *piecewise schedule* from which an AST needs to be generated.

As a next step, we map the tile and point loops to a fixed number of thread blocks and threads. We start by looking for a set of parallel point loops and a set of parallel tile loops. We then strip-mine each loop by the number of thread blocks and threads. For instance, to map a point loop with n iterations to a set of 1024 kernel threads, we strip-mine the loop by a factor of 1024 such that each 1024th iteration is executed by the same thread. The next step is to produce a piece of CPU code that schedules instances of an accelerated kernel, and the kernel code itself that defines the computation of a specific thread in a specific thread block. No actual loops are generated that enumerate the set of thread blocks and threads, but instead the CUDA run-time and hardware spawns a set of blocks and threads and provides the block and thread ID as a parameter to each thread executing the kernel code. To model this, we first generate the outer loop, then we use a *nested context* to introduce the block and thread identifiers and, finally, we generate kernel code that can reference values in the outer CPU code, taking into account the AST generation context of the outer C code as well as the constraints on the kernel and thread identifiers. Exploiting this information is very important to generate high-quality code.

When generating kernel code we also need to rewrite all array subscripts in our compute statement. Traditionally this is done textually by replacing all references to

```
for (c2 = 0; c2 <= 1; c2 += 1)
  for (c3 = 1; c3 <= 4; c3 += 1)
    for (c4 = max(((t1-c3+130) % 128) + c3 - 2, ((t1+c3+125) % 128) - c3 + 3);
         c4 <= min(((c2+c3) % 2) + c3 + 128, -((c2+c3) % 2) - c3 + 134); c4 += 128)
      if (c3 + c4 >= 7 || (c4 == t1 && c3 + 2 >= t1 && t1 + c3 <= 6
                          && t1 + c3 >= ((t1 + c2 + 2 * c3 + 1) % 2) + 3
                          && t1 + 2 >= ((t1 + c2 + 2 * c3 + 1) % 2) + c3)
          || (c4 == t1 && c3 == 1 && t1 <= 5 && t1 >= 4 && c2 <= 1 && c2 >= 0))
        A[c2][6 * b0 + c3][128 * g7 + c4 - 4] = ...;
```

Fig. 1: Copy code from hybrid hexagonal/parallelogram tiling (a single loop)

old induction variables with expressions that compute the values of the old induction variables from new induction variables. When translating an access `A[i+1]` where i now is expressed as $c0 + 1$, a classical rewrite would yield `A[(c0 + 1) + 1]`. With our new approach we represent the expression $i + 1$ itself as a piecewise quasi-affine expression, perform the translation on the piecewise quasi-affine expression, simplify the resulting expression and use our AST generator to *generate an AST expression from this piecewise quasi-affine expression*. As a result we obtain the code `A[c0 + 2]`. In this example the only benefit is increased readability, as any compiler would constant-fold the two additions. However, in general, this concept is a lot more powerful. It allows the specialization of expressions according to the context in which they are generated. If, for instance, an access `A[i == 0 ? N - 1 : i - 1]` is scheduled in a tile where we know i is never 0, we can simplify the access to `A[i - 1]`. This simplification removes the overhead of boundary condition handling from the core computation, a transformation for which a normal compiler misses context information and which traditionally requires specialized statements for boundary and core computations. With our AST generation approach, statements are automatically specialized as soon as boundary computations and core computations are generated as specialized AST subtrees. This is very natural for an AST generator that allows *user-directed versioning*.

After having generated basic CUDA code including the rewritten data accesses, we can start to optimize the code. An essential optimization is to switch from the use of slow “global memory” to the use of fast, manually managed “shared memory“. To do so we need to change the code of each tile such that, before the actual computation takes place, the relevant data from global memory is copied into shared memory, and at the end, the modified data is copied back from shared to global memory. To perform the computation in shared memory, we need to adjust all memory accesses such that they point to the new shared memory arrays and the corresponding locations. How exactly the mapping is computed is outside the scope of this paper, but how we generate the relevant code is interesting. We derive from our mapping a set of piecewise quasi-affine expressions that define the new data locations and *generate AST expressions* for them, relying on the AST generator to ensure that efficient code is generated. This approach enables us to use possibly complex mappings, without writing specialized code generation routines. To create the code that moves the data, we create new statements that copy data from a given global memory location to a given shared memory location and vice versa. In case there is more data to copy than there are threads we use a modulo mapping to assign data locations to threads. Figure 1 shows the code generated to copy data back to global memory. There are various interesting observations possible. First, we see that our modulo expressions have been mapped to the C remainder operator %, which will be translated to fast bitwise operations in case the divisor is a power-of-two constant and the compiler can derive that the dividend is always non-negative. Using the C remainder operator is only possible because we have context information

```
A[0][6 * b0 + 1][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[0][6 * b0 + 2][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 3][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 4][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[1][6 * b0 + 1][128 * g7 + (t1 + 126) % 128) - 2] = ...;
A[1][6 * b0 + 2][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 3][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 4][128 * g7 + (t1 + 126) % 128) - 2] = ...;
```

Fig. 2: Copy code from hybrid hexagonal/parallelogram tiling (unrolled)

about the value of `t1`. Otherwise we would need to fall back to expensive `floord` or `intMod` expressions, dealing with arbitrary (possibly negative) integers, as the state-of-the-art AST generators `CLooG` and `CodeGen+` do. Secondly, we see that we generate a reasonably dense loop nest that enumerates the statements. Because of the presence of existentially quantified variables in the input description, this is by itself non-trivial (see Section 7.3).

Nevertheless, we observe that the generated code is not very efficient. Every loop iteration performs very little computation and evaluates a complex condition. One might hope the condition could be simplified further, but unfortunately the data modified when moving a 5-point stencil forming a cross over a hexagonal tile shape is by itself already non-convex. Applying another level of modulo scheduling makes the necessary compute pattern even more complex, such that obtaining a simpler loop structure is difficult. However, by using *polyhedral unrolling* on the inner three loops and by specializing the statements according to the iteration they are unrolled for, we can remove almost all control overhead. The result is shown in Figure 2. The code is very smooth and each array subscript is specialized to the specific location. We can also see that for the conditionally executed statements the subscripts are optimized according to the conditions such that the remainder operations disappear entirely. Unrolling this code is not trivial, as it needs to be performed in the presence of multiple loop boundaries as well as strides and we need to support the generation of guarded instructions when unrolling. The guarded instructions at the innermost level are very cheap on a GPU, as they can be implemented as predicated instructions. In this small example this is not very visible, but for realistic tile sizes a larger number of statements share the same conditions. We perform similar unrolling for the compute code in our kernel to ensure sufficient instruction level parallelism is available.

The code in Figure 2 is now close to optimal. However, so far we have only looked at a simplified example, a single tile that does not touch any iteration space boundaries. If the iteration space boundaries are taken into account, the generated code becomes a lot more complex. To ensure we can still use the “close to optimal” code most of the time, we use *user directed versioning* to isolate the core computation (the full tiles) from the set of tiles that need to take into account the boundary conditions (partial tiles). Doing so gives us maximal specialization and best performance. However, we now specialize and unroll not only the core computation, but also the code that was introduced to handle the boundary cases, which increases the size of the generated code as well as the time necessary to generate it. When targeting a GPU this may be acceptable, but for FPGAs [Zuo et al. 2013] the cost may be prohibitive. This problem can be easily addressed by using *fine-grained options* to limit the amount of unrolling and specialization in the boundary tiles.

In summary, extending AST generation beyond the creation of control flow makes it possible to use automatic AST generation in complex scenarios. Even though existing AST generators combined with workarounds such as duplicating statements before running the AST generator can be used to solve some of the previously mentioned AST generation issues, such workarounds only exist for some features, they apply only in simple special cases and often inhibit other necessary transformations. By instead carefully integrating several important new extensions into a single AST generation approach, we significantly extend the concept of automatic AST generation such that it is usable in complex AST generation scenarios. We ensure that the different features do not block each other, but when combined provide novel opportunities and solutions to complex AST generation problems. As a result we hope to not only significantly simplify AST generation, but to enable its use in new optimization scenarios.

3. POLYHEDRAL MODEL

The polyhedral model [Feautrier and Lengauer 2011] is a powerful abstraction for analyzing and transforming (parts of) programs that are “sufficiently regular”. The key feature of this model is that it is instance based. That is, each statement instance (i.e., each dynamic execution of a statement inside a loop nest) and each array element is treated individually through the use of a compact representation such as polyhedra [Loechner and Wilde 1997] or Presburger relations [Pugh and Wonnacott 1994]. A program is typically represented using *iteration domains*, containing the statement instances, *access relations*, mapping statement instances to the accessed array element(s), *dependences*, relating statement instances that depend on each other, and a *schedule*, assigning an execution order to the statement instances.

In terms of AST generation, the most relevant elements are the iteration domain and the schedule, where the iteration domain describes the statement instances that need to be executed and the schedule describes the order in which they should be executed. For the iteration domain, we use the representation proposed by Verdoolaege [2011], where each statement instance is represented by a name (identifying the statement) and a tuple of integers (identifying the instance). For each statement, the instances in the iteration domain are described using a *Presburger formula*. We call such a set a *named Presburger set*. Other representations of iteration domains can easily be converted to such a named Presburger set.

Before defining Presburger formulas, let us first consider *affine* expressions, which are terms composed of variables, integer constants, symbolic constants, addition ($+$) and subtraction ($-$). Multiplication by an integer constant is available as syntactic sugar for repeated addition or subtraction. Symbolic constants have a fixed but unknown value and typically represent problem sizes. A *quasi-affine* expression additionally allows integer division by an integer constant ($\lfloor \cdot/d \rfloor$). A Presburger formula is then constructed from quasi-affine expressions, comparison ($\leq$) and the first order logic operators: conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), existential quantification ($\exists$), and universal quantification ($\forall$). A *piecewise quasi-affine* expression is a list of pairs of named Presburger sets and quasi-affine expressions. The sets are pairwise disjoint and the value of the piecewise quasi-affine expression at a given point is equal to the value of the quasi-affine expression associated to the set that contains the point.

Binary relations on pairs of named integer tuples can be defined in a similar way and are called *named Presburger relation*s. Although we will use a more structured representation for schedules in Section 6, it is instructive to consider the basic case of rep-

```
for (int i = 0; i < n; ++i) {
S1: s[i] = 0;
    for (int j = 0; j < i; ++j)
S2:        s[i] = s[i] + a[j][i] * b[j];
S3: b[i] = b[i] - s[i];
}
```

Fig. 3: Example Program

resenting schedules as named Presburger relations proposed by Verdoolaege [2011]. These named Presburger relations associate an integer tuple to each statement instance and the execution order expressed by the schedule is given by the lexicographic order of these integer tuples. Consider for example the program in Figure 3. The iteration domain is

$$\{\, \mathrm{S1}(i) : 0 \le i < n; \mathrm{S2}(i,j) : 0 \le j < i < n; \mathrm{S3}(i) : 0 \le i < n \,\}. \tag{1}$$

One way of expressing the original execution order is the schedule

$$\{\, \mathrm{S1}(i) \to (i,0,0); \mathrm{S2}(i,j) \to (i,1,j); \mathrm{S3}(i) \to (i,2,0) \,\}. \tag{2}$$

That is, the statement instances are first ordered according to the first (or only) element in their instance identifier tuple. Then, for those statement instances that have the same value, the instance of S1 is executed before the instances of S2, which in turn are executed before the instance of S3. Finally, the instances of S2 are executed according to the second element in its instance identifier tuple. For the other two statements all instances are already scheduled apart, so the final schedule coordinate does not need to make a distinction between instances. It may therefore be set to an arbitrary value (here 0).

An alternative execution order may be obtained through the schedule

$$\{\, \mathrm{S1}(i) \to (0,i,0,0); \mathrm{S2}(i,j) \to (1,i,1,j); \mathrm{S3}(i) \to (1,i+1,0,0) \,\}, \tag{3}$$

where all instances of S1 are executed before any instance of S2 or S3. Among the instances of S2 or S3, those where the first dimension of S2 is one more than the single dimension of S3 are executed together, with the S3 instance being executed before the S2 instances.

The purpose of the AST generator is to construct an AST that visits the elements of the iteration domain in the lexicographic order of the integer tuples assigned to the iteration domain elements by the schedule. The construction uses several operations on named Presburger sets and relations available in `isl` [Verdoolaege 2010], including domain and range of a relation, intersection, union, set difference, projection, shared constraints ("simple hull"), simplification of a set (relation) with respect to known constraints ("gist"), integer affine hull and coalescing (replacing pairs of disjuncts by single disjuncts without introducing spurious elements) [Verdoolaege 2015]. All these operations commonly change the elements contained in the sets (relations) they operate on with the exception of coalescing, which only affects the *representation* of a set (relation). Note that the AST generator presented in this paper does *not* use the convex hull operation, as this may introduce constraints with large coefficients.

4. DATA STRUCTURES

The core AST generation algorithm translates schedule constraints into lower and upper bounds of the `for` loops of the generated AST. In order to be able to construct the `for` loops, the algorithm may need to break down the schedule domain into several pieces, resulting in a tree of `for` loops generated from outermost to innermost. There may, however, also be other constraints that cannot be directly encoded in the lower and upper bounds of `for` loops and that need to be generated as `if` conditions instead. In general, we want these conditions to be inserted as high up as possible. On the other hand, we do not want to insert too many redundant conditions, while some conditions may only be redundant with respect to the code that is generated underneath those conditions. Moreover, some constraints, especially disjunctive constraints, may only get discovered at later stages and need to be hoisted up. The AST therefore cannot be constructed in a single pure pre-order depth-first traversal of the schedule. Instead, we perform a single depth-first traversal with some pre-order operations, mainly decomposing the schedule, and some post-order operations, actually constructing the AST. The details of the algorithm will be described in Section 5. In this section, we introduce vocabulary and data structures for passing information up and down the depth-first traversal. These are necessary to describe the current part of the decomposed schedule, the information that is passed down, the actual AST that is being constructed, and the information that is passed up.

4.1. Executed Relation

In the base case of the AST generation algorithm, the schedule is given by a named Presburger relation mapping statement instances to their relative multi-dimensional execution times. The loops in the generated AST are derived from these multi-dimensional execution times. During the AST generation, it is therefore more natural to consider the inverse of this schedule relation, which we call the *executed relation* and which maps execution time vectors to the statement instances that should be executed at those times. For example, given the schedule relation in (3), the (initial) executed relation (without domain constraints) is

$$\{\, (0, s_1, 0, 0) \to \mathrm{S1}(s_1); (1, s_1, 1, s_2) \to \mathrm{S2}(s_1, s_2); (1, s_1, 0, 0) \to \mathrm{S3}(s_1 - 1) \,\}. \tag{4}$$

The levels of the depth-first pass over the schedule correspond to the input dimensions of this executed relation. At each level, the domain of the executed relation is broken up into pieces along that dimension and each piece of the executed relation is considered in turn.

4.2. Stock

The AST node corresponding to a dimension in the domain of the executed relation is constructed upon leaving that level during a depth-first traversal. However, the main information about the AST node is already available when that level is first entered. Some of this information needs to be stored and forwarded through the traversal. We introduce the *stock* to collect this information that can be used to simplify the descendant AST nodes. The stock mainly keeps track of two pieces of information, the conditions on symbolic constants and outer loop iterators that are known to hold at the current position, and a mapping from loop iterators to schedule dimensions. At each level of the depth-first traversal, a new stock is created that is initialized from the stock passed down from the higher level.

The conditions come in two groups, the *generated* conditions and the *pending* conditions. The generated conditions are those for which the algorithm has already decided that they *will* be enforced by the outer nodes in the AST. These are typically the loop bounds on the outer loop nodes. The pending conditions are those that *may* end up being enforced by the outer nodes. They may also get dropped if they turn out to be implied by the inner AST nodes. Note that this distinction is only relevant at the point where the actual AST nodes are being constructed. At other points in the AST generation algorithm, we can simply consider the combination of the two groups of constraints.

The mapping from loop iterators to schedule dimensions is needed because unlike other AST generators, ours exploits the fact that a schedule only specifies a *relative* execution order. The loop iterators in the final AST may then not correspond exactly to the schedule dimensions in the input schedule due to, e.g., scaling or strip-mining by the AST generator, while these schedule dimensions may still be referenced from other parts of the schedule through options (see Section 5.6) or advanced schedule tree nodes (see Section 6).

4.3. Abstract Syntax Tree

The generated AST contains only syntactical information and has been designed to be easily translatable to both C and compiler IR. Each node of the AST is of one of four types, an *if-node*, a *for-node*, a *block-node* or a *user-node*. An if-node has an AST expression as condition, a then-node and optionally an else-node. A for-node has initialization, condition, and increment expressions, and a *body-node*. A block-node represents a compound statement and maintains a list of nodes. Finally, the statement expressed by a user-node is represented as an AST expression.

An AST expression is itself a tree with operators in the internal nodes and integer constants or identifiers as the leaves. The set of operators contains the standard operators found in C-like programming languages, but also higher level operators such as `min` and `max`. Boolean logical operators and the conditional operator (`cond ? a : b`) are available in two forms, one using short-circuit evaluation and one using eager evaluation. We found in our work on low-level compilers [Grosser et al. 2012] that eagerly evaluating operands, instead of using C's short-circuit evaluation, is often beneficial as it reduces control overhead and simplifies the hoisting of loop invariant subexpressions.

The integer division operator also comes in different forms, one of them corresponding to the mathematical operation $\lfloor a/b \rfloor$. Unfortunately, this operation cannot be translated directly into `a / b` in C because the /-operator in C rounds toward zero ([ISO 1999, 6.5.5]) rather than toward negative infinity. A correct translation to C involves a condition on the sign of a, which can bring significant extra costs on some architectures such as GPU devices. We therefore also have a form of the integer division where the result is known to be an integer (such that rounding becomes irrelevant) and one where the dividend is known to be non-negative. The user can specify a preference for these latter forms in which case the AST expression generator will look for opportunities to use them (see Section 5.10). Similarly, the remainder operator comes in two special forms, one where the dividend is known to be non-negative and one where the result of the operations is only compared against zero. In these special cases, the remainder operator can be translated into the %-operator in C.

4.4. Annotated AST

The AST nodes are created after having visited all the children in the depth-first traversal of the schedule. The AST generator may however decide to not encode some of the conditions that need to be satisfied by the symbolic constants in the generated AST nodes such that these conditions may be hoisted up to higher levels. An *annotated AST* keeps track of both the (purely syntactical) AST itself and such extra pieces of polyhedral information. Besides the conditions described above that still need to be enforced by the AST at higher levels, the annotated AST also keeps track of the conditions that *have* been enforced already. This latter set of conditions can then be used to simplify or even eliminate some of the pending conditions in the stock at higher levels.

5. AST GENERATION

This section describes the core AST generation algorithm. In Section 6.3, we will see that this algorithm is applied for each band node in the schedule tree. Our core algorithm is derived from the "Quilleré et al." algorithm [Bastoul 2004; Quilleré et al. 2000], with several significant changes such as isolation, unrolling, `if`-hoisting, the detection of components, shifted strides, and the optimized generation of AST expressions.

5.1. Overview

Example 5.1. Before we delve into the details of the AST generation algorithm, let us first apply it to the simple example of Section 3, where our algorithm essentially coincides with the standard "Quilleré et al." algorithm. In particular, let us consider the schedule of (2). The initial executed relation, including the domain constraints of (1), is

$$\{\,(i,0,0) \to \mathrm{S1}(i) : 0 \le i < n; (i,1,j) \to \mathrm{S2}(i,j) : 0 \le j < i < n; (i,2,0) \to \mathrm{S3}(i) : 0 \le i < n\,\}. \tag{5}$$

The outermost loop of the generated AST is derived from the first dimension in the domain of this relation. A projection onto this first dimension yields

$$\{\,(i) : 0 \le i < n\,\}. \tag{6}$$

In this simple case, we find a set described by a single disjunct and the lower and upper bound of the generated outermost `for` loop can be trivially read off from the constraints. If the projection were described by multiple disjuncts, then we would have to combine them or break them up into disjoint pieces first. Moving on to the second dimension, we would, in principle, find the following projection onto the outer two dimensions

$$\{\,(i,t) : 0 \le i < n \wedge 0 \le t \le 2\,\}. \tag{7}$$

and generate a loop iterating over the values 0 to 2. However, we can see that no instance of $\mathrm{S1}$ is ordered after any instance of $\mathrm{S2}$ or $\mathrm{S3}$ at this level and similarly for $\mathrm{S2}$ with respect to $\mathrm{S3}$. We therefore have a trivial case of our component detection and continue generating an AST for $\mathrm{S1}$, $\mathrm{S2}$ and $\mathrm{S3}$ separately in that order. Note that other AST generators handle this special case through different mechanisms. In each of the components, the second dimension has a fixed value, so no loop needs to be generated. The same is true for the third dimension in the components containing $\mathrm{S1}$ and $\mathrm{S3}$, respectively. The component containing $\mathrm{S2}$ does result in an extra loop, which is handled in the same way as the outer loop.

Algorithm 3 forms the core of the AST generation and creates a (possibly degenerate) `for` AST node for a given schedule dimension after generating AST nodes for the next schedule dimensions through a call to Algorithm 1. The process of creating such a for-node is detailed in Section 5.3. The input is a single-disjunct set corresponding to the current dimension in the schedule domain. The actual schedule domain at this dimension may however consist of several disjuncts. Algorithm 2 takes care of breaking up (or overapproximating) the schedule domain into disjoint single-disjunct pieces and calling Algorithm 3 on each of them. The different ways of breaking up the schedule domain are explained in Section 5.4 and Section 5.5. In Example 5.1, the relevant schedule domains all consist of a single disjunct such that no additional processing is required.

Finally, Algorithm 1 is the main driver that is called (recursively) for each dimension in the schedule space (i.e., the domain of the executed relation). It calls Algorithm 2 after detecting some special cases. In particular, as long as the inner level has not been

ALGORITHM 1: Generate Next Schedule Dimension ("next")

```
Input: — stock
       — executed relation
if at inner level then
  | return terminate(stock, executed)
end
list := ()
foreach sorted component do
  | /* detect shifted strides and record mapping in f, see Section 5.8                */
  | (stock, executed, f) := detect shifts(stock, executed)
  | /* project domain of executed on outer level dimension, see Section 5.2           */
  | domain := project(executed, level)
  | if has isolation domain then
  |   | (before, domain, after, other) = split on isolation(domain)
  |   | list′ := base(stock, executed, before, false)
  |   | list′ += base(stock, executed, domain, true)
  |   | list′ += base(stock, executed, after, false)
  |   | list′ += base(stock, executed, other, false)
  | else
  |   | list′ := base(stock, executed, domain, false)
  | end
  | /* undo schedule modification in case of shifted strides, see Section 5.8        */
  | list += transform(list′, f)
end
return list
```

reached, the algorithm first looks for components as described in Section 5.7. In Example 5.1, three components are detected at the second schedule dimension. In each component, the algorithm checks for shifted strides as explained in Section 5.8, possibly modifying the executed relation if any shifted strides were detected. The domain of the executed relation is then projected onto the outer level dimensions as explained in Section 5.2. In Example 5.1, this projection yields the set $\{ (i) : 0 \leq i < n \}$ (6) at the outer schedule dimension. Finally, if the user specified a piece of the schedule space that needs to be isolated (see Section 5.6), then the algorithm splits the schedule domain into four parts, the part that comes before the isolated part, the isolated part itself, the part that comes after and the part that is incomparable to the isolated part.

When the inner level has been reached, the schedule makes no further distinction between the statement instances in the range of the current executed relation. We therefore generate an AST for each statement separately. Usually, the executed relation will only associate a single statement instance to a given schedule point and we simply create and return a user node. Otherwise, the AST still needs to iterate over the different instances and it can do so in any order. We therefore extend the domain of the executed relation with a copy of the range and continue processing the new dimensions in the domain of the executed relation until the inner level is reached again, in which case the executed relation is guaranteed to have only a single statement instance associated to a given schedule point.

5.2. Local Schedule Domain Constraints

The lower and upper bounds of a `for` loop generated at a given level are derived from the constraints in the domain of the executed relation that involve the current schedule dimension. These constraints may also involve other schedule dimensions, both those corresponding to outer `for` loops and those corresponding to inner `for` loops. Since the

ALGORITHM 2: Generate Component (“base”)

```
Input: — stock
       — executed relation
       — domain: part of schedule domain at current level for which AST should be generated
       — isolated: boolean indicating whether domain refers to isolated part
type := generation type(level, isolated)
if type = unroll then
    (bound, $n$) := find lower bound(domain)
    list := ()
    for $0 \leq i < n$ do
        /* select domain elements at offset i from the lower bound */
        domain′ := slice(domain, bound, $i$)
        domain′ := shared constraints(domain′)
        list += (create(stock, executed, domain′))
    end
    return list
end
if type = separate then
    domain list := separate(domain, executed)
else if type = atomic then
    domain list := (shared constraints(domain))
else
    domain list := make disjoint(domain)
end
list := ()
foreach domain in sorted domain list do
    list += (create(stock, executed, domain))
end
return list
```

ALGORITHM 3: Create for-node (“create”)

```
Input: — stock
       — executed relation
       — bounds: single disjunct set describing bounds on current level for which an AST
         should be generated
domain := bounds intersected with domain of executed
stock′ := detect strides(stock, domain)
stock′ := check for single iteration(stock′, bounds)
list := next(stock′, executed)
/* combine a list of annotated ASTs into a single annotated AST, see Section 5.9 */
list := combine(list, stock, stock′)
return construct for loop(bounds, stock, list)
```

lower and upper bounds of a `for` loop can only refer to iterators of outer loops and (obviously) not to those of inner loops, we first need to project the domain of the executed relation onto its first level dimensions. This operation may introduce additional existentially quantified variables, which cannot be encoded directly in AST expressions. We therefore need to remove them in some way.

One way of removing the existentially quantified variables is to perform quantifier elimination. This process preserves the meaning of the set and therefore ensures that only those values of the current schedule dimension that have any associated statement instances will be executed. On the other hand, quantifier elimination may break up the schedule domain into several pieces. In particular, `isl` performs quantifier elimination by applying parametric integer programming [Feautrier 1988] to compute explicit, quasi-affine representations of the quantified variables. In general, this may split the domain into several parts, each with its own quasi-affine expressions. Other quantifier elimination algorithms lead to similar decompositions of the schedule domain.

Another way of removing the existentially quantified variables is to perform Fourier-Motzkin elimination on them. This may result in an overapproximation of the schedule domain, but it is guaranteed not to break up the schedule domain into several disjuncts. In our AST generator, we take this second option in order to avoid the code size expansion resulting from the first option. That is, we apply Fourier-Motzkin elimination to remove all existentially quantified variables for which we do not have an explicit representation yet. Note that we will only consider the constraints of the possible overapproximation as having been generated at this level, such that any constraint on the actual schedule domain that is not satisfied by this overapproximation will end up getting enforced at a deeper level.

Example 5.2. As an example of a case where these two approaches produce different results, consider the following projection of some schedule domain onto the outermost schedule dimension,

$$\{ (t) : (\exists \alpha : \alpha \geq -1 + t \wedge 2\alpha \geq 1 + t \wedge \alpha \leq t \wedge 4\alpha \leq N + 2t) \}, \tag{8}$$

where N is a symbolic constant. Applying quantifier elimination to the set description in (8) results in

$$\{ (t) : (t \geq 3 \wedge 2t \leq 4 + N) \vee (t \leq 2 \wedge t \geq 1 \wedge 2t \leq N) \}. \tag{9}$$

Note that this is the same set, but that it is described in a different way, without existentially quantified variables. Also note that the description now consists of two disjuncts. Applying Fourier-Motzkin elimination on the α variable in (8) on the other hand results in

$$\{ (t) : 2t \leq 4 + N \wedge N \geq 2 \wedge t \geq 1 \}. \tag{10}$$

This set contains an extra element that is not an element of the set in (8). In particular, it contains the extra element

$$\{ (2) : 2 \leq N \leq 3 \} \tag{11}$$

Figure 4 shows the elements of the set in (8) in black for $0 \leq N \leq 7$. These are the same as those in the set description in (9) that results from quantifier elimination. The additional element (if any) introduced by Fourier-Motzkin (10) is shown in red.

Using the set in (9) would produce the code

```
for (int c0 = 1; c0 <= min(2, floord(N, 2)); c0 += 1)
  // body
for (int c0 = 3; c0 <= floord(N, 2) + 2; c0 += 1)
  // body
```

whereas using the set in (10) produces the code

```
for (int c0 = 1; c0 <= floord(N, 2) + 2; c0 += 1)
  // body
```

As explained above, `isl`'s quantifier elimination may replace a quantified variable by a quasi-affine expression. During the simplification of set descriptions, `isl` may perform a similar substitution. We therefore need to take into account that the projection

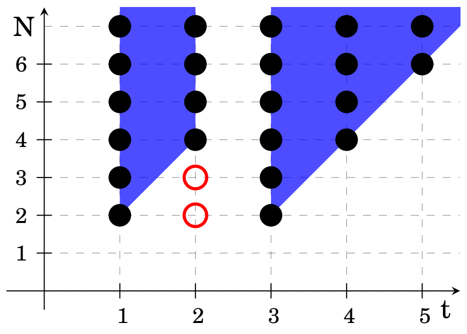

Fig. 4: Quantifier elimination (blue area) and Fourier-Motzkin (black dots + red circles) applied to (8) (black dots).

of the schedule domain, even after eliminating the existentially quantified variables using Fourier-Motzkin elimination, may involve such quasi-affine expressions. If any of these quasi-affine expressions depend on the current level, then they are also eliminated using Fourier-Motzkin elimination since a lower bound of a loop cannot depend on the value of the loop iterator. Similarly, a constraint involving such an expression cannot be used to construct an upper bound since it may fail to hold for some values of the iterator while being satisfied by higher values of the iterator. Note that this process also removes any stride information, but this information will be recovered from the executed relation as explained in Section 5.3.1.

Example 5.3. Consider an iteration domain

$$\{\, \mathrm{S}(i) : 3\left\lfloor \frac{i+1}{3} \right\rfloor \le i \wedge i \ge 0 \wedge i \le 3 \,\} \tag{12}$$

with schedule

$$\{\, \mathrm{S}(i) \to (i) \,\}. \tag{13}$$

The schedule domain and its projection onto the outermost (and only) level is

$$\{\, (i) : 3\left\lfloor \frac{i+1}{3} \right\rfloor \le i \wedge i \ge 0 \wedge i \le 3 \,\}. \tag{14}$$

The constraint $3\lfloor (i+1)/3 \rfloor \le i$ involves a quasi-affine expression in terms of the current level and therefore cannot be used in the construction of the `for` loop bounds. Instead, the expression is eliminated, resulting in the schedule domain

$$\{\, (i) : i \ge 0 \wedge i \le 3 \,\}. \tag{15}$$

The eliminated constraint is then taken into account at the innermost level, resulting in the following code.

```
for (int c0 = 0; c0 <= 3; c0 += 1)
  if ((c0 + 1) % 3 >= 1)
    S(c0);
```

Note that when reaching the innermost level, we can no longer perform any approximations and we have to perform quantifier elimination on any remaining existentially quantified variables. This may then result in a disjunctive condition around the generated statement. Also note that the quantifier elimination procedure of the Omega library is different from the one used by `isl`, but it may also result in splitting the domain. A detailed comparison of the two is beyond the scope of the present article. It is not clear at which point and how far `CodeGen+` applies quantifier elimination, but it appears to be tailored to constraints that only involve a single existentially quantified variable.

5.3. For Node Construction

5.3.1. Stride Detection. Besides the stock and the executed relation, Algorithm 3 for creating a for-node takes as extra input a convex set called "bounds" from which the bounds of the for-node will be extracted. The first step is the detection of strides, but this information is not available in bounds since it does not involve any quasi-affine expressions that depend on the current level. Instead, we use the intersection of the bounds set with the domain of the executed relation. The strides are extracted from the integer affine hull [Verdoolaege 2010] of the resulting set. This operation extracts the equality constraints satisfied by the elements of the set and preserves (most of) the stride information. Let

$$h(\mathbf{p}) + u\,(vi + sf(\boldsymbol{\alpha})) = 0 \tag{16}$$

be one of these equality constraints, with i the current schedule dimension, $\mathbf{p}$ the symbolic constants and outer dimensions and $\boldsymbol{\alpha}$ the existentially quantified variables. Furthermore, v and s have no common factor and f and h are affine expressions such that the coefficients of f have no common factor. If v is nonzero and s is greater than one, then this equality represents a non-trivial stride constraint. Using Bézout's identity $av + bs = 1$, we can rewrite (16) to

$$ui = -ah(\mathbf{p}) + us\,(bi - af(\boldsymbol{\alpha})) \tag{17}$$

so that $-ah(\mathbf{p})$ is a multiple of u and i is equal to $o(\mathbf{p}) = -ah(\mathbf{p})/u$ modulo s. The offset $o(\mathbf{p})$ and the stride s are stored in the stock. If there is more than one equality with a non-trivial stride, then the offsets and strides can be combined and the overall stride will be the least common multiple of the strides. In particular, if we have two offset/stride pairs

$$i = o_1(\mathbf{p}) + s_1\,f_1(\boldsymbol{\alpha}) \tag{18}$$
$$i = o_2(\mathbf{p}) + s_2\,f_2(\boldsymbol{\alpha}) \tag{19}$$

then let g be the greatest common divisor of s_1 and s_2 and let c and d be such that $c\,s_1 + d\,s_2 = g$ (Bézout's identity once more). Multiplying the equation in (18) by $t_1 = d\,s_2/g$ and the equation in (19) by $t_2 = c\,s_1/g$, we obtain

$$i = (d\,s_2 + c\,s_1)/g\,i = t_1 o_1(\mathbf{p}) + t_2 o_2(\mathbf{p}) + (s_1\,s_2)/g\ (bf_1(\boldsymbol{\alpha}) + af_2(\boldsymbol{\alpha}))\,. \tag{20}$$

That is, the combined offset is $t_1 o_1(\mathbf{p}) + t_2 o_2(\mathbf{p})$, while the combined stride is $(s_1\,s_2)/g$, the least common multiple of s_1 and s_2.

Example 5.4. Consider the schedule domain

$$\{\,(i) : \exists \alpha, \beta : 0 \le i \le 100 \wedge n - i + 6\alpha = 0 \wedge m - i + 10\beta = 0\,\}, \tag{21}$$

with n and m symbolic constants. For the constraint $n - i + 6\alpha = 0$, we have, using the notation of (16), $h(\mathbf{p}) = n$, $u = 1$, $v = -1$, $s = 6$ and $f(\boldsymbol{\alpha}) = \alpha$. We may take $a = -1$ and $b = 0$ to find $o_1(\mathbf{p}) = n$ with $s_1 = 6$. We similarly find $o_2(\mathbf{p}) = m$ and $s_2 = 10$. We have $g = 2$ and may take $c = 2$ and $d = -1$, resulting in a combined stride of 30 and a combined offset of $-5n + 6m$. To see that this combined offset satisfies both stride constraints, note that the combination of the original two stride constraints implies that $n - m$ is a multiple of two. That is, the bounds set (from which the existentially quantified variables have been eliminated) is of the form

$$\{ (i) : 0 \leq i \leq 100 \wedge 2 \left\lfloor \frac{m-n}{2} \right\rfloor = m - n \}. \tag{22}$$

We therefore have that $m - (-5n + 6m) = 5(n - m)$ is indeed a multiple of 10.

5.3.2. Loop Constraints. After detecting the strides, we add the constraints enforced by the `for` loop generated for the current schedule dimension to the stock that will be used in the construction of descendant nodes such that these constraints may be used to simplify those descendant nodes. The actual construction of the `for` loop corresponding to the current schedule dimension is only performed after these descendant nodes have been created. At that point we will need to make a distinction between the constraints that are enforced by the bounds of the `for` loop and the constraints that may need to be enforced by an extra `if` conditional. We therefore take this difference into account while updating the stock, even though this distinction has no influence on the descendant nodes.

In order to determine which constraints are enforced by the `for` loop corresponding to the current schedule dimension, we first need to know if we are even going to construct a `for` loop. In particular, based on the constraints in the current stock, the constraints in bounds and the stride constraints (if any), we may be able to determine that the current schedule dimension can attain only a single value. In this case, we will generate a special "degenerate" `for` loop which we allow the user to translate to an assignment of the initial value to the loop iterator. Note that this single value will in general be specified as a piecewise quasi-affine expression in the symbolic constants and the outer loop iterators. If it turns out that this expression consists of a single quasi-affine expression, then we do not generate any `for` loop at all, but instead substitute the current schedule dimension for this single quasi-affine expression in the executed relation. We refer to this case as an eliminated `for` loop. The reason for only performing this substitution when the single value is described by a single quasi-affine expression is that otherwise we would be introducing additional disjuncts in the executed relation. In the eliminated case, we eliminate the current schedule dimension from bounds and add the result to the pending constraints. In the other cases, we add the constraints in bounds that do not involve the current schedule dimension to the pending constraints and we add the remaining constraints in bounds as well as the stride constraint (i.e., the fact that the schedule dimension is equal to the offset plus a multiple of the stride) to the generated constraints.

Example 5.5. Consider the schedule domain

$$\{ (i) : i \geq 1 \wedge n - 1 \leq i \leq n \wedge 4 \left\lfloor \frac{i-2}{4} \right\rfloor = i - 2 \}, \tag{23}$$

where n is a symbolic constant. The stride constraint $4 \lfloor (i-2)/4 \rfloor = i - 2$ does not appear in bounds, but it is added back for the purpose of looking for a single value. From these constraints, we can see that i attains a single value of $4 \lfloor (n+2)/4 \rfloor - 2$ and that this value is represented as a single quasi-affine expression. Substituting this value in the schedule domain, we obtain

$$\{ () : n \geq 2 \wedge \left\lfloor \frac{n}{4} \right\rfloor \leq n - 2 \}. \tag{24}$$

These constraints are then added as guards to the annotated AST at the innermost level.

Note that degenerate loops that are not also eliminated are fairly rare in practice. We only report our handling of this situation for completeness, but it may very well not be close to optimal. In most of the remaining rare cases, the loop could in fact be eliminated, but we simply fail to derive an appropriate single quasi-affine expression. In fact, these cases used to occur more frequently, but most of these have been resolved through an improved detection of single quasi-affine expressions.

After generating an annotated AST for the body of the for-node, we know which constraints on the symbolic constants and outer loop iterators are enforced by this subtree and we can (optionally) use them to simplify the pending constraints. Additionally, the pending constraints are simplified with respect to the generated constraints. The simplified pending constraints are then combined with the constraints hoisted from the annotated AST for the body (see Section 5.9) and an additional set of *implied* constraints. The implied constraints are those constraints that are implied by the generated constraints, but that may not be implied by their AST expression counterparts. In particular, these are the constraints implied by the stride constraints. In case of a degenerate loop, this also includes the constraints implied by the generated constraints. Since we are allowing the user to consider a degenerate `for` loop as an assignment, the fact that the upper bound is greater than or equal to the lower bound (i.e., that there even is a single iteration of the loop) is not enforced by this assignment and therefore needs to be considered separately. The combination of the simplified pending constraints, the hoisted constraints and the implied constraints will then definitely be generated either at the current level or hoisted up to a higher level. From this point on, in particular for the construction of the AST expressions for the `if` conditions that have not been hoisted out of the body and those for the bounds of the `for` loop, they may therefore be considered as generated constraints.

Example 5.6. As an example of the effect of exploiting enforced constraints, consider the iteration domain

$$\{\, \mathrm{S}(i,j) : 0 \leq i < m \wedge 0 \leq j < n \,\}, \tag{25}$$

where m and n are symbolic constants, with schedule

$$\{\, \mathrm{S}(i,j) \to (i,j) \,\}. \tag{26}$$

Projection of the schedule domain onto the outer dimension yields

$$\{\, (i) : 0 \leq i < m \wedge n \geq 1 \,\}. \tag{27}$$

The single pending constraint at this level is therefore $n \geq 1$. At the inner level, the schedule domain, simplified with respect to the stock constraints, is

$$\{\, (i,j) : 0 \leq j < n \,\}. \tag{28}$$

The `for` loop generated at this inner level enforces the constraint $n \geq 1$ so it can optionally be used to simplify the pending constraint at the outer level. If we exploit this enforced constraint, we generate the code

```
for (int c0 = 0; c0 < m; c0 += 1)
  for (int c1 = 0; c1 < n; c1 += 1)
    S(c0, c1);
```

Otherwise, the pending constraint is turned into an `if` condition and we generate the code

```
if (n >= 1)
  for (int c0 = 0; c0 < m; c0 += 1)
    for (int c1 = 0; c1 < n; c1 += 1)
      S(c0, c1);
```

Example 5.7. As an example of constraints implied by a stride constraint, consider the schedule

$$\{ \mathrm{S}(t) \to (t) : \exists \alpha : 2t - n = 4\alpha \wedge 0 \leq t \leq 100 \}, \tag{29}$$

with n a symbolic constant. Stride detection finds a stride of 2 and an offset of $n/2$. The stride constraint $(n/2 - t) \bmod 2 = 0$ is encoded as

$$n - 2t - 4 \left\lfloor \frac{n + 2t}{4} \right\rfloor = 0. \tag{30}$$

This constraint implies that n is a multiple of 2. Since this stride constraint is added to the generated constraints, this fact may be simplified away at the deeper levels. However, it is not implied by the actually generated `for` loop. We therefore eliminate t from (30) and add the resulting constraints to the constraints that need to be generated at the outer level. The final code is

```
if (n % 2 == 0)
  for (int c0 = (n / 2) + 2 * floord(-n - 1, 4) + 2; c0 <= 100; c0 += 2)
    S(c0);
```

Note that we have exploited the fact that n is a multiple of 2 while generating the loop initialization.

5.3.3. AST Expressions. Let us now consider the construction of the initialization and the condition of the generated `for` loop from the lower and upper bounds on the current schedule dimension. Since the bounds set may be an overapproximation of the schedule domain, it may in rare cases *not* involve lower and/or upper bounds. If they are missing, then we derive a single piecewise quasi-affine bound from the domain set using parametric integer programming [Feautrier 1988]. If this set does not have a lower bound, then an error is reported. If it has no upper bound, then an infinite for-node is generated. In the standard case where there is one or more lower bound constraint $h(\mathbf{p}) + vi \geq 0$ with $v > 0$, each of the constraints is converted to a lower bound $\ell(\mathbf{p}) = \lceil -h(\mathbf{p})/v \rceil$ and the lower bound on the for-node is set to the maximum (as an AST expression) of these lower bounds. If the loop is strided, however, then we need to make sure that this lower bound has the right value modulo the stride. We therefore first replace each of the lower bounds $\ell(\mathbf{p})$ by $o(\mathbf{p}) + s \lceil (l(\mathbf{p}) - o(\mathbf{p}))/s \rceil$.

Example 5.8. Consider the iteration domain

$$\{ \mathrm{S1}(i) : 0 \leq i \leq M; \mathrm{S2}() \}, \tag{31}$$

where M is a symbolic constant, with schedule

$$\{ \mathrm{S1}(i) \to (i, 0); \mathrm{S2}() \to (0, 1) \}. \tag{32}$$

Assume that at the outer level, we want to generate a single loop for both statements, as explained in Section 5.4, with bounds set $\{ (i) : i \geq 0 \}$. This bounds set does not have any upper bound on the current schedule dimension so we consider the schedule domain

$$\{ (0); (i) : 0 \leq i \leq M \} \tag{33}$$

instead. From this set, we can derive the upper bound

$$\begin{cases} 0 & \text{if } M \leq 0 \\ M - 1 & \text{otherwise.} \end{cases} \tag{34}$$

The generated code is as follows.

```
for (int c0 = 0; c0 <= (M <= 0 ? 0 : M); c0 += 1) {
  if (M >= c0)
    S1(c0);
  if (c0 == 0)
    S2(0);
}
```

Example 5.9. Continued from Example 5.4. The bounds set in (22) has only a single lower bound on the current schedule dimension, $i \geq 0$. The default `for` loop initialization would therefore be $\max\{\lceil -0/1 \rceil\} = 0$. This value may however not satisfy the stride constraint, depending on the values of m and n. It is therefore replaced by

$$-5n + 6m + 30 \left\lceil \frac{5n - 6m}{30} \right\rceil = -5n + 6m + 30 \left\lfloor \frac{5n - 6m + 29}{30} \right\rfloor . \tag{35}$$

Depending on a user setting, the for-node upper bound condition is constructed either as a single comparison of the loop iterator to a minimum of upper bounds, derived analogously to the lower bounds, or as a conjunction of comparisons, each derived directly from an upper bound constraint. The single upper bound is expected by the OpenMP support of some compilers while the conjunction is more efficient on FPGAs [Zuo et al. 2013]. Finally, the independent constraints are added to the annotated AST holding the for-node.

Example 5.10. Consider upper bounds of the form

$$M \geq c_3 + 1 \wedge c_1 \geq 3c_3 + 8 \tag{36}$$

If the user has selected the generation of a single upper bound, then an upper bound condition of the form `c3 < min((c1 + 1) / 3 - 2, M)` is generated, while in the other case, an upper bound condition of the form `M >= c3 + 1 && c1 >= 3 * c3 + 8` is generated.

5.4. Separation

The schedule domain(s) computed in Algorithm 1 may be arbitrary Presburger sets, which in `isl` are represented in disjunctive normal form. The creation of a for-node in Algorithm 3, however, takes a single-disjunct set as input. The responsibility of the intermediate Algorithm 2 is then to replace the schedule domain by an ordered sequence of disjoint single-disjunct domains. There are essentially two ways to obtain such single-disjunct domains, either the entire domain is approximated by a single-disjunct domain or the domain is broken up into single-disjunct parts. The first option may require the introduction of additional guards in the descendants of the currently constructed for-node, possibly leading to run-time overhead. The second option can lead to code duplication since different instances of the same domain may end up in different single-disjunct schedule domain parts.

For each schedule dimension, the user can specify which of these options to take. Alternatively, the user may also specify that the schedule dimension should be unrolled (Section 5.5) or she may leave the option unspecified, in which case the schedule domain is broken up into a list of single-disjunct domains in a pragmatic way.

Let us consider the two main options in some more detail. If the "separate" option is specified, then the schedule domains are computed for each statement separately. Each of these schedule domains is broken up into disjoint single-disjunct sets and a common refinement is computed. This is the standard separation of the "Quilleré et al." algorithm [Quilleré et al. 2000; Bastoul 2004]. In the "atomic" case, the *shared constraints* are used. That is, the constraints of the disjuncts are considered in turn and only those are kept that are satisfied by the entire schedule domain. This process may in some cases result in the absence of a lower and/or upper bound, a case we discussed in Section 5.3.

Example 5.11. Continued from Example 5.8. Consider the schedule domain at the outer dimension in (33), repeated here

$$\{ (0); (i) : 0 \leq i \leq M \}. \tag{37}$$

In case of separation, we consider the schedule domains for each statement separately, i.e., $\{ (0) \}$ and $\{ (i) : 0 \leq i \leq M \}$, and apply the standard separation algorithm, breaking up these sets into disjoint single-disjunct sets, resulting in

$$\{ (0) : M \leq -1 \}, \quad \{ (0) : M \geq 0 \} \quad \text{and} \quad \{ (i) : 1 \leq i \leq M \}. \tag{38}$$

The generated code is as follows.

```
if (M <= -1) {
  S2(0);
} else {
  S1(0);
  S2(0);
  for (int c0 = 1; c0 <= M; c0 += 1)
    S1(c0);
}
```

In the atomic case, we consider the constraints shared by the disjuncts in (33). In this example, there is only one such constraint, i.e., $i \geq 0$. Note that the equality constraint $i = 0$ in the first disjunct is equivalent to $i \geq 0 \wedge i \leq 0$. The generated code for this case is shown in Example 5.8.

5.5. Unrolling

Unrolling in the AST generation works by taking slices of the schedule domain for successive values of the current schedule dimension and by calling "create" for each of these slices. By construction, the schedule dimension has a fixed quasi-affine value in each of the slices and no actual for-node will be created. Two factors play an important role in unrolling: stride detection and the selection of the most appropriate lower bound. Stride detection is performed as explained in Section 5.3.1. If any stride is found, then it is substituted ($i = o(\mathbf{p}) + si'$) in the schedule domain for the purpose of selecting a lower bound.

Example 5.12. If the schedule domain is of the form

$$\{ i : 0 \leq i < 1024 \wedge i \bmod 256 = 0 \}, \tag{39}$$

then it is replaced by $\{ i' : 0 \leq i' < 4 \}$.

The lower bound identification requires a single disjunct so we consider once more the shared constraints of the schedule domain, although in this case we also allow constant shifts of the constraints. For each lower bound constraint $h(\mathbf{p}) + vi \geq 0$ with $v > 0$, we compute the maximum value of $i + 1 - \lceil -h(\mathbf{p})/v \rceil$ over the schedule domain. If the maximum exists and has value n then we know we can cover the schedule domain with at most n slices of the form $i = \lceil -h(\mathbf{p})/v \rceil + t$ with $0 \leq t < n$. We take the lower bound with the smallest such n. If no suitable lower bound can be found, then we report an error.

Example 5.13. For the schedule domain

$$\{ i : 0 \leq i < 1000 \wedge N \leq i < N + 4 \}, \tag{40}$$

we would prefer the lower bound N (with 4 slices) over the lower bound 0 (with 1000 slices).

5.6. Isolation

Isolation in Algorithm 1 is controlled by an option specified by the user. If set, the option describes a part of the schedule domain that should be isolated from the other parts of the schedule domain. The typical use cases are the isolation of full tiles from partial tiles or the isolation of a vectorizable loop from its prologue and epilogue. The atomic/separate/unroll option can be specified separately for the isolated part and the rest of the schedule domain. For any given level, the isolated domain is first projected onto the first level dimensions as in Section 5.2. In particular, the inner dimensions are projected out and all existentially quantified variables and all quasi-affine expressions involving the current dimension are eliminated. We subsequently replace the set by its shared constraints to ensure that the center part is contiguous. An intersection with the current schedule domain yields the center part of the isolation. The "before" part is obtained by first constructing a set of iterations that are executed before some iteration in the center part and then subtracting that center part. Similarly, the "after" part is obtained by first constructing a set of iterations that are executed after some iteration in the center part and then subtracting both the center part and the "before" part. The "other" part is obtained by subtracting the before, center, and after parts from the current schedule domain and consists of those iterations that are incomparable to the center part. If any atomic/separate/unroll option is specified for the rest of the schedule domain, then it is applied to the before, after, and other part separately.

Example 5.14. Assume we have an iteration domain

$$\{\, \mathrm{S}(i) : m \leq i < n \,\}, \tag{41}$$

where m and n are symbolic constants, with initial schedule

$$\{\, \mathrm{S}(i) \to (i) \,\} \tag{42}$$

and that we want to strip-mine the loop by 4, a factor that has been derived to allow the backend compiler to vectorize the inner loop, e.g., by considering the data types used in the computation or the target vector instruction set. We first modify the schedule to

$$\{\, \mathrm{S}(i) \to (4\left\lfloor \frac{i}{4} \right\rfloor, i) \,\} \tag{43}$$

and then we want to single out those iterations that result in an inner loop of exactly four iterations. In particular, we need to make sure that the first schedule dimension $t = 4\lfloor i/4 \rfloor$ belongs to the schedule domain of the second dimension and that so does $t + 3$. We therefore isolate the values of the first schedule dimension that satisfy

$$\{\, (t) : m \leq t \wedge t + 3 < n \,\}. \tag{44}$$

Projecting out inner dimensions and replacing the set by its shared constraints does not modify the isolated set. The before part is

$$\{\, (t) : n \geq 4 + m \wedge t \leq m - 1 \,\}, \tag{45}$$

the after part is

$$\{\, (t) : n \geq 4 + m \wedge t \geq n - 3 \,\} \tag{46}$$

and the other part is

$$\{\, (t) : m - 3 \leq t \leq n - 1 \wedge n \leq m + 3 \wedge n \geq m + 1 \,\}. \tag{47}$$

The generated code (shown below) consists of a single loop performing the prologue computation, two—now easily vectorizable—loops that enumerate the center part, and one loop that forms the epilogue computation. There is also an additional loop nest for the other part, which is executed in case the values of `n` and `m` yield an empty center part. In certain cases, it is possible to avoid the generation of dedicated code for the other part by instead enumerating the relevant iterations as part of the before and after parts. We currently do not perform such an optimization.

```
{
  if (n >= m + 4)
    for (int c1 = m; c1 <= 4 * floord(m - 1, 4) + 3; c1 += 1)
      S(c1);
  for (int c0 = 4 * floord(m - 1, 4) + 4; c0 < n - 3; c0 += 4)
    for (int c1 = c0; c1 <= c0 + 3; c1 += 1)
      S(c1);
  if (n >= m + 4 && 4 * floord(n - 1, 4) + 3 >= n) {
    for (int c1 = 4 * floord(n - 1, 4); c1 < n; c1 += 1)
      S(c1);
  } else if (m + 3 >= n)
    for (int c0 = 4 * floord(m, 4); c0 < n; c0 += 4)
      for (int c1 = max(m, c0); c1 <= min(n - 1, c0 + 3); c1 += 1)
        S(c1);
}
```

Example 5.15. We also briefly illustrate isolation on multiple dimensions. Assume we have an iteration domain

$$\{\, \mathrm{S}(i,j) : 0 \le i < n \wedge 0 \le j < m \,\}, \tag{48}$$

where n and m are symbolic constants, with initial schedule

$$\{\, \mathrm{S}(i,j) \to (i,j) \,\}, \tag{49}$$

which we want to tile to improve register reuse. To implement such a register tiling (e.g., by 3×4), we modify the schedule to

$$\{\, \mathrm{S}(i,j) \to (3 \left\lfloor \frac{i}{3} \right\rfloor, 4 \left\lfloor \frac{j}{4} \right\rfloor, i, j) \,\}. \tag{50}$$

We then want to single out those iterations that result in two inner loops with together exactly 12 iterations, aiming to generate loops that can be fully unrolled without introducing conditions that hinder the use of registers. To implement this, we isolate the tiles of the schedule dimensions $t_3 = i, t_4 = j$ that lie entirely in the schedule domain $\{\, (t_3,t_4) : 0 \le t_3 < n \wedge 0 \le t_4 < m \,\}$. In particular, the tile at $t_1 = 3\lfloor t_3/3 \rfloor, t_2 = 4\lfloor t_4/4 \rfloor$ belongs to the schedule domain if both (t_1,t_2) and $(t_1{+}2, t_2{+}3)$ satisfy these constraints. We therefore isolate the values of the first two schedule dimensions that satisfy

$$\{\, (t_1,t_2) : 0 \le t_1 \wedge t_1 + 2 < n \wedge 0 \le t_2 \wedge t_2 + 3 < m \,\}. \tag{51}$$

The generated AST for this example (shown below) consists of two loop nests at the outer level. The first iterates over tiles that are complete in the first dimension, while the second iterates over tiles that are partial in this dimension. At the second level inside the first loop nest, the tiles are further split into those that are complete and those that are partial in the second dimension. Note that when generating the AST below, we have specified in the input (through a context node, see Section 6) that n and m are known to not be very small ($n > 2 \wedge m > 3$). Without this extra piece of information, the AST generator would generate additional code for such small values of n and m, corresponding to the “other” parts of the isolation.

```
{
  for (int c0 = 0; c0 < n - 2; c0 += 3) {
    for (int c1 = 0; c1 < m - 3; c1 += 4)
      for (int c2 = c0; c2 <= c0 + 2; c2 += 1)
        for (int c3 = c1; c3 <= c1 + 3; c3 += 1)
          S(c2, c3);
    if ((m - 1) % 4 <= 2)
      for (int c2 = c0; c2 <= c0 + 2; c2 += 1)
        for (int c3 = -((m - 1) % 4) + m - 1; c3 < m; c3 += 1)
          S(c2, c3);
  }
  if ((n - 1) % 3 <= 1)
    for (int c1 = 0; c1 < m; c1 += 4)
      for (int c2 = -((n - 1) % 3) + n - 1; c2 < n; c2 += 1)
        for (int c3 = c1; c3 <= min(m - 1, c1 + 3); c3 += 1)
          S(c2, c3);
}
```

5.7. Components

It is important to understand that the schedule only defines a *relative* execution order of statement instances with respect to other statement instances and that the generated loops do not need to correspond exactly to the schedule dimensions. In some cases, we can generate much simpler code by taking a closer look at this relative execution order.

Example 5.16. As a simple example, take an iteration domain

$$\{ S_0(); S_1(i) : 0 \leq i < 10 \} \tag{52}$$

with schedule

$$\{ S_0() \to (0); S_1(i) \to (i) \}. \tag{53}$$

The schedule domains for the two statements separately are $\{ (0) \}$ and $\{ (i) : 0 \leq i < 10 \}$. The latter is also equal to the combined schedule domain. Applying separation would therefore split off iteration 0 from S_1, while using an atomic domain would generate a single loop with a condition on S_0 for the iterator to be equal to 0. A closer look at the schedule reveals, however, that $S_0()$ is not scheduled *after* any iteration of S_1, so that we can simply generate code for S_0 first, resulting in

```
S0();
for (int i = 0; i < 10; ++i)
    S1(i);
```

In the general case, we construct a graph with as nodes the statements and an edge between node a and node b if any instance of b is scheduled after any instance of a and if these instances are mapped to the same value of the outer schedule dimensions. The strongly connected components in this graph may be scheduled in topological order. The above property is evaluated by considering each of the schedule dimensions in turn. If the value for b is always smaller, then the property does not hold. If it may be greater, then the property holds. Otherwise, if the value may be the same, then we continue to the next schedule dimension. If we reach the inner level, then we assume that the instances may be scheduled in any order.

5.8. Shifted Stride Detection

The stride detection of Section 5.3 only works if there is a consistent stride across the entire schedule domain. In some cases, the schedule domains for individual statements may be strided, while the combined schedule domain is not.

Example 5.17. Take an iteration domain

$$\{ A(i) : 0 \leq i < 10; B(i) : 0 \leq i < 10 \} \tag{54}$$

with schedule

$$\{ A(i) \to (2i); B(i) \to (2i+1) \}. \tag{55}$$

Both schedules generated by Feautrier's algorithm [Feautrier 1992] and shifted linear schedules [Darte et al. 2001] may be of this form. While the schedule domains per statement have stride 2 (with offsets 0 and 1), the combined schedule domain is simply $\{ (t) : 0 \leq t < 20 \}$. If we replace the above partial schedule by the equivalent $\{ A(i) \to (2i, 0); B(i) \to (2i, 1) \}$, then the schedule domain in the outer dimension becomes $\{ (t) : 0 \leq t < 20 \wedge t \bmod 2 = 0 \}$ and the stride can be exploited.

To be able to apply the transformation illustrated in the example, we look for a statement where the current schedule dimension does not have an obviously fixed value. Note that it does not make sense to look for any strides if all of them have a fixed value. We then consider pairs of statements where the first is the selected statement and the second is any statement. If for all of these pairs we find that the *differences* between the values of the schedule dimension are of the form $m_i x + r_i$ with m_i and r_i constant values and $m_i > 1$, then we take the greatest common divisor m of the m_i and reduce the r_i modulo m. If m is greater than one and not all of the r_i are equal, then we apply the transformation $(\ldots, t, \ldots) \to (\ldots, t - r_i, r_i, \ldots)$ to the schedule domain of statement i. Since this transformation changes the dimension of the schedule domain, it needs to be undone in the resulting annotated ASTs so that they can be combined with other annotated ASTs.

Example 5.18. Continued from Example 5.17. The differences between the values of the schedule dimension for B and A in (55) are $\{ (t) : -17 \leq t \leq 19 \wedge t \bmod 2 = 0 \}$, such that $m_1 = 2$ and $r_1 = 1$. For pairs of A instances, we similarly find $m_0 = 2$ and $r_0 = 0$.

Note that this transformation is mainly meant to make our AST generator generate nice code even when presented with schedules from outside users that contain such hidden strides. The scheduler that comes with `isl` also implements a Feautrier style scheduler, but it will detect and adjust such shifted strides during the scheduling itself. On the other hand, the Pluto style scheduler [Bondhugula et al. 2008] of `isl` may occasionally construct a schedule with several *distinct* shifted strides. Detecting them during scheduling is therefore complicated, but they may still be detectable in the AST generator since the detection is applied on different components separately. Furthermore, the transformation can also be applied if the schedule itself does not contain an explicit scaling factor but if instead the scaling can be derived from the iteration domain.

5.9. Combining Annotated ASTs

In several parts of the AST generator, we need to combine a list of annotated ASTs into a single annotated AST. This single annotated AST may be used in the same or in an outer stock. In particular, we may want to move to a stock of an outer loop, in which case the extra loop iterator needs to be projected out from both the guard and the enforced condition attached to the ASTs. The shared constraints in these projections are hoisted to the single annotated AST and the original guards are simplified with respect to the hoisted guard. Note that if there is more than one element in the list of annotated ASTs, then some of the constraints may not be explicitly available in the guards of each element. We therefore first intersect the guards with both the stock domain and the enforced constraints of each individual annotated AST. The shared constraints are then still selected from the original guards, but they only need to satisfy the intersections to be considered for hoisting. In general, the shared constraints form a single disjunct set. However, as a special case, if all annotated ASTs in the list have the same guard, then we allow this guard to be hoisted even if it is described in terms of multiple disjuncts.

Example 5.19. Assume that at the outer level, we have decided to construct a `for` loop with bounds $\{ (t) : 1 \leq t \leq 8 \}$. Assume furthermore that at the inner level, we have constructed three annotated ASTs with the following guards (g_i) and enforced constraints (f_i).

$$\begin{aligned} g_1 &= \{ (t) : t \leq 6 \} & g_2 &= \{ (t) : t \geq 2 \} & g_3 &= \{ (t) : t \geq 2 \} \\ f_1 &= \{ (t) : t \geq 2 \} & f_2 &= \{ (t) : t \leq 5 \} & f_3 &= \{ (t) : t \leq 8 \}. \end{aligned} \tag{56}$$

If we only consider the guards g_i themselves, then we cannot hoist any guards since $t \leq 6$ is not satisfied by g_2 and g_3 while $t \geq 2$ is not satisfied by g_1. After intersection with the stock domain and the corresponding enforced constraints, we find

$$g_1' = \{ (t) : 2 \leq t \leq 6 \} \quad g_2' = \{ (t) : 2 \leq t \leq 5 \} \quad g_3' = \{ (t) : 2 \leq t \leq 8 \}. \tag{57}$$

Considering the original two constraints $t \leq 6$ and $t \geq 2$ once more, we now find that $t \geq 2$ is satisfied by all three g_i' and that we may therefore hoist this constraint, which can then be used to simplify the original guards to

$$g_1'' = \{ (t) : t \leq 6 \} \quad g_2'' = \{ (t) \} \quad g_3'' = \{ (t) \}. \tag{58}$$

The simplified guards (if non-trivial) then need to be expressed as if-nodes around the corresponding AST node. We do not, however, treat every annotated AST in the list separately. Instead, we incrementally build up a tree of if-nodes, keeping track of a list of if-nodes that can still be extended without changing the execution order. For each of these if-nodes, we keep track of the condition enforced by the node and its parents as well as the condition enforced by the corresponding else-branch. For each annotated AST in the original list, we look for the deepest if-node such that one of the attached conditions is implied by the current guard. The guard is then simplified in terms of this condition and the AST node (with an enclosing if-node if needed) is inserted in the corresponding branch. Finally, the list of if-nodes is updated to reflect the insertion.

Example 5.20. Assume an outer generated domain of

$$D = \{ (i_0, i_1) : 1 \leq i_0 \leq u \wedge i_0 \leq i_1 \leq n \wedge 1 \leq l \leq u \leq n \}, \tag{59}$$

with l, u and n symbolic constants, and a list of four annotated ASTs with the following simplified guards.

$$\begin{aligned} g_1 &= \{ (i_0, i_1) : l \leq i_0 \wedge i_0 + 1 \leq i_1 \} && (\texttt{s0}) \\ g_2 &= \{ (i_0, i_1) : i_0 < l \} && (\texttt{s3}) \\ g_3 &= \{ (i_0, i_1) : l \leq i_0 \wedge i_0 + 1 \leq i_1 \wedge u < n \} && (\texttt{s2}) \\ g_4 &= \{ (i_0, i_1) \} && (\texttt{s1}) \end{aligned} \tag{60}$$

When we consider g_1, the list of if-nodes is still empty, so we create a first if-node with condition $c_0 = g_1$ and complement $\bar{c}_0 = D \setminus g_1$, and add the graft to the final list of grafts. Guard $g_2 \cap D$ is not a subset of c_0, but it is a subset of $\bar{c}_0$ ($i_0 < l$ is satisfied by the complement of $l \leq i_0$). We therefore simplify g_2 in the context of $\bar{c}_0$, which has no effect in this case, and extend the else-branch of this if-node with the second graft. We also extend the list of if-nodes with condition $c_1 = g_2$ and complement $\bar{c}_1 = \bar{c}_0 \setminus g_2$. The following guard $g_3 \cap D$ is a subset of neither c_1 nor $\bar{c}_1$, but it is a subset of c_0. We therefore simplify g_3 in the context of c_0, which yields $g_3' = \{ (i_0, i_1) : u < n \}$ and extend the then-branch of the if-node with the third graft. Since we are extending the first element in the list of if-nodes, we drop all subsequent elements in the list of if-nodes so that they can no longer be extended. This ensures that this combination step never changes the order of the statements, even though it may be slightly too conservative. We also extend the list of if-nodes with a new second if-node with condition $c_1' = g_3'$ and $\overline{c_1'} = c_0 \setminus g_3'$. The final guard $g_3 \cap D$ is a subset of none of the if-node guards or complements. We therefore clear the list of if-nodes and add the graft directly to the final list of grafts. The generated code then has the following form.

```
if (c0 >= lb && c1 >= c0 + 1) {
  s0(c0, c1);
  if (n >= ub + 1)
    s2(c0, c1);
} else if (lb >= c0 + 1)
  s3(c0, c1, lb, c0, c1);
for (int c3 = max(lb, c0); c3 <= ub; c3 += 1)
  s1(c0, c1, c3);
```

5.10. Generating AST Expressions

Most of the operations performed by the AST generator are performed on `isl` data types that represent Presburger sets and relations or piecewise quasi-affine functions. It is only during the final construction of the if- and for-nodes that these objects are converted to syntactic AST expressions. Internally in `isl`, quasi-affine functions are expressed in terms of greatest integer parts ($\lfloor\cdot\rfloor$). In principle, these expressions can be translated directly into their AST expression counterparts, but as explained in Section 4.3, for some use cases it is important to know if the first argument of an integer division is non-negative or if the division is exact. Moreover, we typically want an expression of the form $m \lfloor (a(\mathbf{i})/m) \rfloor$ to be translated to $a(\mathbf{i}) - (a(\mathbf{i}) \bmod m)$, provided again that $a(\mathbf{i})$ is non-negative.

Whenever generating an `if` or `for` condition or a `for` initialization or upper bound expression from an expression involving greatest integer parts, we first check for opportunities to extract modulo expressions and then check the signs of the remaining greatest integer parts. Note that when generating a conjunction of constraints, we first generate expressions for the constraints not involving greatest integer parts and then add those constraints to the stock so that they can be used to simplify the remaining constraints. It does not matter in which order these conditions get *evaluated* in the generated code, but we do need to consider a particular order to ensure that there are no cycles in constraints being used to simplify other constraints. The functions for generating AST expressions from `isl` objects are also available to the user and can for example be used to generate an AST expression from an access function.

Example 5.21. Consider the code in Figure 7b. The constraints for the loop epilogue are of the form $n \geq 2 \wedge n - 4\lfloor n/4 \rfloor \geq 2$. We first generate a condition for the constraint $n \geq 2$ so that it can be used to simplify the condition generated from $n - 4\lfloor n/4 \rfloor \geq 2$.

If we are generating an equality constraint, we first check if the equality encodes a stride. If so, the stride can be expressed in the AST using an expression of the form `x % m == 0`. In this case, the sign of `x` is of no importance. For other constraints or expressions in general, if we find a subexpression of the form $fm\lfloor a(\mathbf{i})/m \rfloor$ and we can prove that $a(\mathbf{i})$ is non-negative based on the constraints in the stock, then the expression is replaced by $f \cdot a(\mathbf{i}) - f \cdot (a(\mathbf{i}) \bmod m)$. If $a(\mathbf{i})$ may be negative, but $-a(\mathbf{i}) + m - 1$ can be proved to be non-negative, then it is replaced by $f \cdot (m+1-a(\mathbf{i})) - f \cdot ((m+1-a(\mathbf{i})) \bmod m)$ instead, exploiting the fact that $\lfloor a/b \rfloor = -\lceil -a/b \rceil = -\lfloor (-a+b-1)/b \rfloor$.

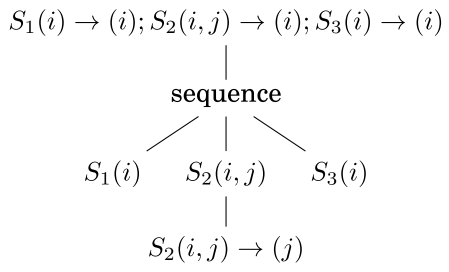

Fig. 5: Schedule tree representation of the original schedule (2)

Moreover, the $a(\mathbf{i})$ inside the argument of mod can be replaced by any $a'(\mathbf{i}) = a(\mathbf{i}) + me(\mathbf{i})$. We therefore look for constraints $h(\mathbf{i}) \geq 0$ among the shifted shared constraints of the context with coefficients that are either equal or opposite to those of $a(\mathbf{i})$ modulo m. Since $h(\mathbf{i})$ is known to be non-negative in the context, it can be used directly as $a'(\mathbf{i})$. If no such constraint can be found, we check if $a(\mathbf{i})$ or $-a(\mathbf{i}) + m - 1$ themselves can be proved to be non-negative by solving an ILP problem. The latter test is also used to check if the first arguments of the remaining integer divisions are non-negative. These simple heuristics appear to work out fairly well in practice.

Example 5.22. Continued from Example 5.21. The constraint $n - 4\lfloor n/4 \rfloor \geq 2$ has a subexpression of the form $fm\lfloor a(\mathbf{i})/m \rfloor$, with $f = 1$, $m = 4$ and $a(\mathbf{i}) = n$. Since we have added the constraint $n \geq 2$ to the generated constraints of the stock, we can take $h(\mathbf{i}) = n-2$. We may therefore replace $4\lfloor n/4 \rfloor$ by $n-(n \bmod 4)$ and obtain the constraint $n \bmod 4 \geq 2$.

6. SCHEDULE TREES

This section describes our schedule representation that generalizes on other schedule representations such as named Presburger relations and explains how to adjust the AST generation algorithm to take such a schedule tree as input.

6.1. Motivation

Schedules used in polyhedral compilation naturally have the form of a tree. This is clearly the case of schedules that represent the original execution order of a program, as they deal with loops and compound statements. In particular, a compound statement first executes all statement instances in the first "macro statement" (which may itself be a loop or a compound statement) and then all statement instances in the next macro statement. This order can be expressed by a "sequence" node that expresses that its children should be executed in order. The execution order of a loop can be expressed as an affine function that expresses the loop iterator (or its negative) in terms of the statement instances. The statement instances are then executed in increasing order of this affine function. For example, a schedule tree representation of the original execution order (2) is shown in Figure 5.

Similarly, automatic scheduling algorithms [Darte et al. 2001; Bondhugula et al. 2008] typically construct a schedule recursively. At each level of the recursion, the algorithms analyze the remaining dependences and determine which statements need to be scheduled together and how these groups of statements need to be scheduled with respect to each other (either in sequence or in an arbitrary order). Each group of statements is then scheduled according to a partial schedule, which is typically an affine function and which may be multi-dimensional in case of algorithms that look

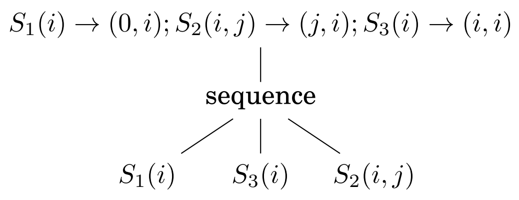

Fig. 6: Schedule tree computed by `isl` for the program in Figure 3

for tiling opportunities [Bondhugula et al. 2008]. Within each group, the dependences are then updated to only include those dependences that are not carried by the partial schedule and the algorithm proceeds to the next level of the recursion. For example, the schedule tree in Figure 6 is a possible outcome of such an automatic scheduling algorithm.

Previously proposed generic schedule representations such as Kelly's abstraction [Kelly and Pugh 1995], "$2d+1$"-schedules [Girbal et al. 2006] and named Presburger relations [Verdoolaege 2011] are essentially an *encoding* of such schedule trees. While such encodings are in principle sufficient as input to an AST generator, they make intermediate operations such as tiling specific nodes in the schedule tree much more cumbersome. By taking schedule trees as input to our AST generator, we allow the user to construct an initial schedule tree (either corresponding to the original execution order or obtained from an automatic scheduler), perform various manipulations on the tree such as affine transformations of partial schedules (including interchange, skewing, reversal), statement reordering, fusion, distribution, index set splitting, strip-mining and tiling and then send the result to the AST generator without having to convert the schedule tree to another representation. For more details on how these operations are performed on schedule trees and a comparison to other schedule representations we refer to Verdoolaege et al. [2014].

The additional node types allow the user to express that a collection of groups of statements may be executed in any order and that some symbolic constants are only available in specific subtrees of the schedule tree. They also allow the user to introduce extra statements relative to the current position in the schedule tree. These features are of crucial importance for a clean implementation of a polyhedral compiler such as PPCG [Verdoolaege et al. 2013]. PPCG takes a sequential C program as input, applies an automatic scheduler to detect and exploit parallelism and tiling opportunities, and then maps the resulting schedule tree to a CUDA or OpenCL device. If the outer node in the result of the scheduler identifies groups of statements that can be executed in any order, then PPCG can map each of these groups independently, possibly mapping some of them to the CPU rather than to the device. The other groups may also have some of the outermost nodes mapped to the CPU and it is therefore important to be able to express that the symbolic constants corresponding to the block and thread identifiers (using CUDA terminology) of the device are only available in the subtrees that are actually going to be executed by the device. Additional statements for synchronization and for copying data between global memory and shared memory or registers can be inserted directly at the appropriate positions in the schedule tree. Although it is technically possible to write a tool such as PPCG without schedule trees,[4] it requires much more effort to get right, it is more cumbersome to extend and specialize

[4]In fact, the initial implementation of PPCG was built on top of an earlier version of our AST generator taking only named Presburger relations as input [Verdoolaege et al. 2013].

for target-specific optimizations, and may involve nested calls to the AST generator, which are more difficult to understand and debug.

6.2. Nodes

Besides the iteration domain and the schedule, an AST generator usually also takes a third piece of information as input, namely a set of constraints on the symbolic constants called the *context*. This context can be used to simplify expressions in the generated AST. In our schedule trees, the context and iteration domain are included as additional outer nodes, requiring only a single object as input. It should be noted though, that although these nodes are included in the same schedule tree object, the information about which statement instances need to be executed (available in the domain node) and in which order they need to be executed (available through most of the other nodes) is still kept separate. Another reason for including these nodes in the schedule tree is that context nodes are also allowed in other positions of the schedule tree, where they express information that cannot be expressed in any other commonly used schedule representation.

The following types of nodes are available in our schedule trees. We only consider the properties that are relevant to AST generation.

Domain. A domain node appears as the root of a schedule tree and introduces the statement instances that are scheduled by the descendants of the domain node. The statement instances are described by a named Presburger set.

Context. A context node introduces symbolic constants and known constraints on those symbolic constants. The introduced symbolic constants can be used in the descendants of the context node. Guards involving the symbolic constants are not allowed to be hoisted outside of the AST that corresponds to the subtree of the context node. The context node typically appears as the child of the root of the schedule tree, but additional context nodes in the tree may also be useful to introduce symbolic constants that are only available in specific subtrees, e.g., block and thread identifiers in PPCG. Symbolic constants referenced by the root domain node do not need to be introduced by a context node.

Filter. A filter node selects a subset of the statement instances introduced by outer domain, expansion or extension nodes and retained by outer filter nodes. Filter nodes are typically used as children of set and sequence nodes (described next), where the siblings select the other statement instances. Filters can also be used to select the statement instances that are mapped to a given value of a symbolic constant introduced in an outer context node. For example, PPCG uses filter nodes to select the instances mapped to a given block or thread identifier.

Sequence. A sequence node expresses that its children should be executed in order. These children must be filter nodes, with mutually disjoint filters.

Set. A set node is similar to a sequence node except that its children may be executed in any order.

Band. A band node contains a partial schedule on the statement instances introduced by outer domain, expansion or extension nodes and retained by outer filter nodes. This partial schedule may be piecewise quasi-affine, but is required to be total on those statement instances. Additionally, a band node contains options that control the AST generation, e.g., separation (Section 5.4), unrolling (Section 5.5) or isolation (Section 5.6). The ability to specify AST generation options for each band separately allows for great flexibility for more complicated programs with statements that, after scheduling, are spread over different, possibly nested, loop nests.

Each band node contains an unstructured schedule of the type accepted by the AST generator in Section 5

Leaf. A leaf of the schedule tree. Leaf nodes do not impose any ordering.

Mark. A mark node can be used to attach any kind of information to a subtree of the schedule tree.

Expansion. An expansion node maps each of the domain elements that reach the node to one or more domain elements. The image of this mapping forms the set of domain elements that reach the child of the expansion node. Expansion nodes are useful for grouping instances of several statements into an instance of a single virtual statement to ensure that those instances are always executed together.

Extension. An extension node instructs the AST generator to add additional domain elements that need to be scheduled. The additional domain elements are described in terms of the outer schedule dimensions. A typical use case of such extension nodes is the introduction of data copying statements for locality optimization. This is expressed as a relation between the outer schedule dimensions and the set of array elements accessed by the statement instances scheduled to a give value of those schedule dimensions. The new statement instances of extension nodes are only introduced if any of the instances introduced in outer nodes may still be executed at the given position of the schedule tree.

Guard. A guard node describes constraints on the symbolic constants and the schedule dimensions of outer bands that need to be enforced by the outer nodes in the generated AST. This allows the user to rely on a given set of constraints being enforced at a given point in the schedule tree, independently of the simplification of guards performed by the AST generator.

Sequence and set nodes have one or more children. A leaf node has no children. All other types of nodes have exactly one child.

6.3. AST Generation

AST generation for a schedule tree performs a depth-first traversal of the schedule tree, where Algorithm 1 is applied on each band node. Termination of this algorithm is replaced by a visit to the child of the band node in the schedule tree, while the original termination is performed at the leaves of the schedule tree. The handling of the other node types is fairly straightforward. Sequence and set nodes simply handle their children in order and combine the resulting annotated ASTs. Filter nodes and context nodes impose the given constraints on either the range of the executed relation or (a prefix of) its domain. When annotated ASTs are combined on leaving a context node, any symbolic constants introduced by the context node are projected out from the guard and the enforced condition. Guard nodes add the associated conditions to the conditions that still need to be enforced by the AST at higher levels. Expansion nodes expand the range of the executed relation, while extension nodes add extra elements to this relation.

The core algorithm needs to be slightly adjusted to the presence of descendant nodes in the schedule tree. In particular, the shifted stride detection of Section 5.8 is skipped if any of the descendants (or in fact the current band node itself) refers to the schedule dimensions of outer nodes, i.e., context, guard and extension nodes or band nodes with an isolation domain. For the band nodes that are nested inside other band nodes, the isolation domain is expressed in terms of the schedule dimensions of both the current and the outer band nodes. The reason for skipping shifted stride detection is that the schedule space is modified on a per statement basis, such that there is no way to express the mapping between the original and the transformed schedule space. Other changes to the schedule space are taken into account through the mapping from loop iterators to schedule dimensions in the stock.

Likewise, the detection of components of Section 5.7 needs to take into account the descendants of the current node, while checking if an instance of statement b is scheduled after any instance of statement a. In particular, if the child of the band node is a sequence node and if a and b are mapped to different children, then their order determines whether the property holds. If the child is a set node and a and b are mapped to different children, then the property does not hold since the instances of a and b may be scheduled in any order with respect to each other. As long as the property has not been evaluated (and therefore some instances of a and b may be scheduled together), the traversal of the schedule tree continues. When reaching a leaf node, the instances may be scheduled in any order. Note that it is important that the entire schedule tree is available for the AST generator to be able to evaluate this property. Otherwise, the AST generator would have to conservatively assume that the property holds for instances that are co-scheduled by the known part of the schedule.

7. EXPERIMENTAL RESULTS

Let us now consider quantitative aspects of our AST generator, evaluating how it fares compared to the state of the art. The version of `isl` used in these experiments is `isl-0.14-368-g23e8573`.

7.1. Robustness

One of the major goals of our AST generator is that it should accept any well-formed input and that it should produce correct code. In order to test the correctness of our AST generator, we collected inputs from the `CLooG` and `CodeGen+` distributions and verified that we produce correct code for each of them. Since the types of inputs of these tools form a strict subset of the inputs accepted by our AST generator, the inputs can be easily converted to schedule trees with three nested nodes, a domain, a context and a band node. To verify the correctness of the output, we parse the output using `pet` [Verdoolaege and Grosser 2012] and verify that the statement instances executed by the output are the same as those specified by the input and that they are executed in an order that matches the input schedule. Moreover, if the input schedule is single-valued (which is usually the case), then we check that each statement instance is executed exactly once. We perform this test for various settings of the options that control the shape of the output. Although we are aware of work that checks that different versions of `CLooG` produce equivalent output [Verdoolaege et al. 2012], we are not aware of prior work that systematically verifies that the generated AST matches the input schedule.

Performing the same check on `CodeGen+` generated output from `CLooG` inputs, we find that for 13 test cases (out of 94) `CodeGen+` produces output containing `N/A`. This means that `CodeGen+` was unable to express the statement instance in terms of the loop iterators in the scheduled code, possibly due to `CodeGen+` applying the schedule as a preprocessing step and doing the AST generation on the schedule domain, which prevents the recovery of the statement instance in case of non-injective schedules. We also found one test case (`darte`) where the generated code contains a condition on a loop iterator outside of the loop over which it iterates and one test case (`walters`) where some statement instances in the input do not appear in the generated code.[5] Note that the `N/A` issue also appears in four of the (disabled) test cases that come with the `CodeGen+` distribution. One of the original `codegen` test cases (`p.delft2`) also produces the error “guard condition too complex to handle”. Performing a similar test

[5]These issues were reported to the authors in October 2013.

```
if (n >= 2)
  for (i = 2; i <= n; i += 2) {
    if (i%4 == 0)
      S0(i);
    if ((i+2)%4 == 0)
      S1(i);
  }
```

(a) CLooG 0.18.1 generated code

```
for (int c0 = 2; c0 < n - 1; c0 += 4) {
  S1(c0);
  S0(c0 + 2);
}
if (n >= 2 && n % 4 >= 2)
  S1(-(n % 4) + n + 2);
```

(b) isl generated code

```
#define intMod(a,b) ((a) >= 0 ? (a) % (b) : (b) - abs((a) % (b)) % (b))
for(i = 2; i <= n; i += 2)
  if (intMod(i,4) == 0)
    S0(i);
  else
    S1(i);
```

(c) CodeGen+ generated code

Fig. 7: Code for example from [Chen 2012, Figure 8(b)] (style edited)

```
for(i=1; i<=n-2; i++) {
  S0(i,i);
  S1(i,i);
  for(j=i+1; j<=n-1; j++)
    S1(i,j);
  S1(i,n);
  S2(i,n);
}
S0(n-1,n-1);
S1(n-1,n-1);
S1(n-1,n);
S2(n-1,n);
S0(n,n);
S1(n,n);
S2(n,n);
for (i=n+1; i <= m; i++)
  S3(i,j);
```

(a) CLooG 0.14.1

```
for(i=1; i<=m; i++) {
  if(i>=n +1) {
    S2(i,n);
  } else {
    S0(i,i);
    S1(i,i);
    if (i>=n)
      S2 (i,i);
  }
  for(j=i+1; j<=n-1; j++)
    S0(i,j);
  if(n >= i+1) {
    S0(i,n);
    S2(i,n);
  }
}
```

(b) `CodeGen+`

```
for (int c0=1;c0<=n;c0+=1) {
  S0(c0, c0);
  for (int c1=c0;c1<=n;c1+=1)
    S1(c0, c1);
  S2(c0, n);
}
for (int c0=n+1;c0<=m;c0+=1)
  S2(c0, n);
```

(c) isl codegen

Fig. 8: Code for youcefn taken from [Bastoul 2004, Figure 6] (style edited)

on `CLooG` with `CodeGen+` inputs is not feasible since older versions of `CLooG` (prior to our enhancements) would not allow existentially quantified variables in the input.

7.2. Generated Code Quality

We illustrate the improvements in code quality of our AST generator through examples from related work. Figure 7 compares the outputs on an input with iteration domain $\{\, \texttt{S0}(i) : \exists\alpha : 1 \le i \le n \wedge i = 4\alpha; \texttt{S1}(i) : \exists\alpha : 1 \le i \le n \wedge i = 4\alpha + 2 \,\}$ and schedule $\{\, \texttt{S0}(i) \to (i); \texttt{S1}(i) \to (i) \,\}$, an example reconstructed from Chen [2012, Figure 8(b)] with the iteration space extended to negative numbers. Notice that thanks to the shifted stride detection of Section 5.8, the modulo operation is removed from the inner loop in the `isl` output. During the construction of the `n % 4 >= 2` condition, we exploit the fact that the body is only executed if the condition `n >= 2` also holds, as explained in Section 5.10. For the input of Chen [2012, Figure 8(a)], `isl` produces the same AST as `CodeGen+`.

Figure 8c illustrates the detection of components of Section 5.7. The example is the one used by Bastoul [2004, Figure 6]. Without the detection of components, some separation is needlessly applied as illustrated in Figure 8b for `CodeGen+` and in Figure 8a for `CLooG` 0.14.1. The code of Bastoul [2004, Figure 7] is similar to the code that `isl` produces, but was obtained either manually or using a preliminary implementation of an "unisolate" technique that was never made public. As a further illustration, for the `darte` input, each statement only occurs twice in the `isl` output even with full separation. In the `CLooG` output, each statement occurs 5 times, and is surrounded by several modulo conditions, some of which are redundant. As explained in Section 7.1, `CodeGen+` produces incorrect output in this example.

Even though the C snippets just presented already allow the reader to understand certain code properties, the instruction sequence that is executed on the target may noticeably differ from the C code we see and often depends very much on the target compiler used. To understand how well the above ASTs can be understood and optimized by regular C compilers, we analyze the size of the generated code as well as the number of instructions executed when running this code and combine it with an analysis of the generated assembly code. For this experiment, we compile the code with clang 3.6, gcc 4.9.1 and icc 15.0.0 using the options `-O3 -std=gnu99 -march=corei7-avx -mtune=corei7-avx -fno-inline`. In additional compilations, we disable vectorization (`-fno-vectorize`, `-fno-tree-vectorize`, and `-no-vec`) and loop unrolling (`-fno-unroll-loops`, `-unroll=0`). The size of the generated code is measured as the size of the function it is located in, which we obtain by calling `nm -S` on the object file. The number of dynamic instructions is measured with valgrind's callgrind tool, a profiling tool that traces the execution of the code and counts for each function the number of instructions executed. When measuring the dynamic instruction count we do not aim for a cycle-accurate performance prediction, but want to understand the different compilers and their optimizations without getting lost in target specific details.

Table I shows the code size and instruction counts we obtain for the code in Figure 7 and Figure 8. As statements we use `A[x]++` and `B[x]++` for Figure 7 while for Figure 8, we use `A[x][y]++`, `B[x][y]++` and `C[x][y]++`. For the code in Figure 7 we see that with clang the `isl` generated code is larger than the `CLooG` and `CodeGen+` generated code, due to the loop epilogue `isl` generates. Clang is able to exploit the knowledge that i is positive to only retain the positive branch of `CodeGen+`'s `intMod` instruction, but it cannot remove the `if` conditions that `CLooG` and `CodeGen+` introduce in the loop body. Similarly, gcc also simplifies the `intMod` instruction and leaves the conditional control flow in the loop. As the `isl` generated code does not evaluate conditions in the loop body, it executes notably fewer instructions (around 190 instructions) than the `CodeGen+` and `CLooG` generated code (355 – 506 dynamic instructions). Neither clang nor gcc unroll or vectorize any of the generated code. icc, on the other hand, vectorizes and unrolls the code. In case we disable both, icc does not succeed in eliminating the negative branch of `intMod()` and also does not transform the remainder computation to a bitwise *and* (i.e., `i%(2^n)` into `i&(2^n-1)`). As a result, the icc generated code for `CodeGen+` and `CLooG` is not only noticeably larger but also requires 878 or even 1105 dynamic instructions, compared to the 192 instructions executed when running the code icc derives for the `isl` generated AST. In case we allow unrolling, icc successfully eliminates the `intMod` branch, but still fails to express the remainder as a bitwise *and*, a transformation that gcc and clang apply in all important cases. When enabling automatic vectorization, icc derives vector code only for the `isl` generated AST. Vectorization induces a large code size increase (240 bytes to 1104 bytes), but slightly reduces the instruction count. A larger decrease of the dynamic instruction count is not possible as the memory access pattern of our test case is strided and memory accesses on AVX require additional scalar loads or shuffle instructions. Instruction sets that allow gather and scatter instructions are likely to show more visible benefits from vectorization. The same holds for loop kernels that contain more complex statements, where the increased complexity of the memory accesses has a lower relative impact.

The performance results we obtain for youcefn in Table I show that with unrolling and vectorization disabled, the `isl` AST generator has a clear code-size benefit across all compilers, with little difference visible in the dynamic instruction count. When allowing loop unrolling, icc is the only compiler that exploits it and as a result reduces the dynamic instruction count across all ASTs, with the largest effects visible for `isl` and `CodeGen+`. Noticeably, the `isl` generated code now only executes 16598 instructions requiring only 784 bytes in code size. When additionally enabling vectorization, the dynamic instruction count of all compilers is reduced even more significantly: the `isl` generated code compiled with icc yields 10489 dynamic instructions, which is very close to the 10383 instructions yielded by `CLooG` compiled with icc, while its size is only 1008 bytes compared to 2960 bytes for the latter (one third of the code size for a similar dynamic instruction count).

Wrapping up this code quality study, `isl` can perform control flow optimizations that notably improve performance, complementing loop nest transformations exposing parallelism and locality. These optimizations are not performed by normal C compilers, and conversely, some of them enable further compiler optimizations such as automatic vectorization. The code generated by `isl` is also significantly smaller in general. Note that the latter may open opportunities for further unrolling and vectorization for a given code size budget.

		Code Size [Byte]			**Inst. Count**		
		clang	gcc	icc	clang	gcc	icc
Figure 7	`CLooG` 0.18.1	74	89	272	506	356	497
	`CodeGen+`	**68**	**67**	**256**	381	355	471
	`isl`	125	113	1104	**193**	**188**	**188**
Figure 7 - no vector	`CLooG` 0.18.1			272			497
	`CodeGen+`			256			471
	isl			**240**			**182**
Figure 7 - no vector - no unrolling	`CLooG` 0.18.1			**112**			1105
	`CodeGen+`			160			878
	isl			**112**			**192**
youcefn	`CLooG` 0.14.1	801	1057	2960	16260	13412	**10383**
	`CodeGen+`	**552**	914	2160	15548	14235	10594
	isl	625	**812**	**1008**	**13845**	**13409**	10489
youcefn - no vector	`CLooG` 0.14.1	398	443	1632	26853	**26271**	20928
	`CodeGen+`	245	289	1280	27266	26868	17322
	isl	**181**	**218**	**784**	**26762**	26570	**16598**
youcefn - no vector - no unrolling	`CLooG` 0.14.1			464			27738
	`CodeGen+`			400			**27679**
	isl			**208**			31917

Table I: Code size and dynamic instruction count for Figure 7 and Figure 8. (with increment in stmts)

```
// Simple
S(n % 128);

// Shifted
S(((t1 + 121) % 128) + 7);

// Conditional
if ((t1 + 121) % 128 <= 123)
  S(((t1 + 125) % 128) + 3);
```

(a) isl

```
// Simple
for(i = intMod(n,128); i <= 127; i += 128)
  S(i);

// Shifted
for(i = 7+intMod(t1-7,128); i <= 134; i += 128)
  S(i);

// Conditional
for(i = 7+intMod(t1-7,128); i <= 130; i += 128)
  S(i);
```

(b) `CodeGen+`

Fig. 9: Modulo conditions (examples not supported by `CLooG`)

```
// Two e.q. variables
for (int c0 = 0; c0 <= 7; c0 += 1)
  if (2 * (2 * c0 / 3) >= c0)
    S(c0);

// Multiple bounds
for (int c0 = 0; c0 <= 1; c0 += 1)
  for (int c1 = max(t1 - 384, t2 - 514);
       c1 < t1 - 255; c1 += 1)
    if (c1 + 256 == t1 ||
        (t1 >= 126 && t2 <= 255 && c1 + 384 == t1) ||
        (t2 == 256 && c1 + 384 == t1))
      S(c0, c1);
```

(a) isl

```
// Two e.q. variables
S(0); S(2); S(3);
S(4); S(5); S(6); S(7);

// Multiple bounds
if (t1 >= 126)
  S(0, t1 - 384);
S(0, t1 - 256);
if (t1 >= 126)
  S(1, t1 - 384);
S(1, t1 - 256);
```

(b) isl unrolled

Fig. 10: Existentially quantified variables (examples not supported by `CLooG`/`CodeGen+`)

7.3. Modulo Mappings and Existentially Quantified Variables

Our algorithm aims to generate a valid and efficient AST for any Presburger relation. Dealing with existentially quantified variables is one area not sufficiently covered in Section 7.1. These can result from modulo mappings from global to shared memory, or from a full iteration space to a set of thread identifiers. Indeed, generating an efficient AST in the presence of existentially quantified variables requires particular care with modular arithmetic, eliminating remainders of integer divisions, or simplifying them whenever possible. We start with a simple modulo operation $\{ S[i] \to [i] : i = n \bmod 128 \}$ (in a context where $n \geq 0$) to verify that modulo operations can be detected at all. Since older versions of `CLooG` (prior to our enhancements) do not allow existentially quantified variables, we do not compare against it in this section. For `isl` and `CodeGen+`, Figure 9 shows that `isl` uses a single statement with a remainder operation, whereas `CodeGen+` generates a loop. Using a loop is very inefficient due to the call to `intMod` and the additional control flow overhead. However, it can be optimized by observing that the expression `n%128` is an invariant of all surrounding loops, which should be accessible to the loop-invariant code motion pass of a compiler. Two slightly more complicated examples are mappings from a set of iterations to a set of threads $t1$ with $0 \leq t1 < 128$. The first mapping is the one-to-one mapping $\{S[i] \to [i] : 7 \leq i \leq 134 \land i \bmod 128 = t1\}$ which `isl` again translates into a single instruction, the second is the mapping $\{S[i] \to [i] : 7 \leq i \leq 130 \land i \bmod 128 = t1\}$

		Code Size [Byte]			Inst. Count		
		clang	gcc	icc	clang	gcc	icc
Modulo – simple	`CodeGen+`	100	66	400	15	5	42
	isl	38	28	32	9	8	8
	isl (unsigned mod)	**12**	**11**	**16**	**3**	**3**	**3**
Modulo – shifted	`CodeGen+`	116	145	416	14	15	43
	isl	45	33	48	10	9	8
	isl (unsigned mod)	**16**	**16**	**32**	**4**	**4**	**4**
Modulo – conditional	`CodeGen+`	116	145	416	14	15	43
	isl	78	63	80	19	18	18
	isl (unsigned mod)	**29**	**29**	**32**	**8**	**8**	**8**
Two e.q. variables	isl	49	49	64	8	8	8
	isl unrolled	49	49	**48**	8	8	**6**
Multiple bounds	isl	260	367	160	1966	2120	3392
	isl unrolled	**140**	**150**	**112**	**20**	**21**	**20**

Table II: Code size and dynamic instruction count for Figure 9 and Figure 10

which maps 124 iterations to 128 threads. `isl` lowers this mapping to a single conditional statement. `CodeGen+` generates for both cases a full loop nest. It is interesting to note that all previously shown loops are degenerate loops with just a single iteration. `CodeGen+` is not able to detect those loops, whereas `isl` is designed to always recognize degenerate loops (see Section 5.3).

For the previous test cases only a single existentially quantified variable was introduced due to the single modulo operation in the schedule. For more complex use cases, e.g., a modulo mapping of access functions that already contain modulo expressions or nested modulo mappings, it is often possible that multiple existentially quantified variables are introduced. The first test case $\{S[i] \to [i] : \exists(\alpha, \beta : i = 2\alpha + 3\beta \wedge 0 \leq \alpha < 3 \wedge 0 \leq \beta \wedge 0 \leq i < 8)\}$ involves two existentially quantified variables in a single equality. `CodeGen+` aborts here with the message *guard condition too complex to handle*. In Figure 10 we see that `isl` is able to generate valid code (see Section 5.2 for details), which can be unrolled both for better efficiency and to better understand the computation that is performed. The next test case is $\{S[i,j] \to [i,j] : \exists(\alpha, \beta : 0 \leq i \leq 1 \wedge t1 = j + 128\alpha \wedge 0 \leq j + 2\beta < 128 \wedge 510 \leq t2 + 2\beta \leq 514 \wedge 0 \leq 2\beta - t2 \leq 5)\}$, which was reduced from the example in Figure 1. `CodeGen+` aborts with *Can't generate multiple wildcard GEQ guards right now*. `isl` either generates a loop with multiple loop bounds or, if unrolled, a set of conditional statements. As Chen [2012] does not discuss how existentially quantified variables are handled, the scope of support in `CodeGen+` is unclear. When inspecting the source code of `CodeGen+` we found several code paths that require a single existentially quantified variable per constraint. `isl` has no limitations on the number of existentially quantified variables per constraint (see Section 5.2).

Using the approach presented in Section 7.2 we again analyze the target code generated by different compilers for the examples above. The results in Table II show for the different modulo conditions both a code size and especially a dynamic instruction count benefit of `isl` over `CodeGen+`, with the icc generated code being exceptionally bad for `CodeGen+`. One exception is the code for the conditional modulo, where `CodeGen+` uses slightly fewer dynamic instructions (14 vs. 19 for clang) due to the `isl` generated code evaluating two modulo expressions, one in the condition and one in the statement itself. It is interesting to note that a single modulo expression is expensive even though it uses a power-of-2 divisor which should be lowered to an efficient bitwise *and* operation. Indeed, none of the compilers tested has sufficient information to perform this optimization when using signed types. As we know for this example that the dividend of the remainder is always non-negative, we can transfer the necessary information by manually casting the modulo dividends to unsigned integer types. `isl` marks remainder instructions with a non-negative dividend and even actively forms them (see Section 5.10), such that the insertion of the relevant casts can be automatized, if desired. As is clear from the results, the knowledge about the positivity of the dividend allows all compilers to optimize the modulo conditions such that the `isl` generated code across all compilers and all modulo test cases is now both notably smaller and requires fewer dynamic instructions than the code produced by `CodeGen+`.

The kernel with two existentially quantified variables results in almost identical target code for all compilers, independently of it being unrolled or not. As the size of the loop is statically known and very small, all compilers unroll the loop automatically. In contrast, the more complex loop in the "multiple bounds" test case is not unrolled by any compiler, but instead a complex piece of code with a high number of dynamic instructions is generated. By taking advantage of the context information available in the AST generator we can unroll the code to an efficient sequence of (in parts predicated) statements. This reduces code size and results in a large reduction of the dynamic instruction count.

Overall, `isl` makes measurable progress in generating more efficient target code from general Presburger relations with existentially quantified variables. This ability allows a better decoupling between the exploration of complex loop transformations on one side, and the (re)generation of efficient control flow on the other. `isl` achieves this by leveraging state-of-the-art optimizations for modular arithmetic, complementing these with specific attention to the sign of operands and actively forming expressions that have non-negative dividends. `isl` also leverages integer arithmetic to identify cases where a loop is not needed, simplifying the generation of compact control flow. Finally, while for loops with constant trip and small instruction count, loop unrolling can be performed by the C compiler, partial unrolling of more complex loops is preferably performed by an AST generator due to the additional context information available. This is confirmed and pushed further in these `isl` experiments.

7.4. Index Set Splitting

Traditionally, when optimizing a loop program by changing the execution order of individual statement instances, the order of the different instances is described by a schedule that contains for each statement a single quasi-affine expression which assigns execution times to statement instances. For certain transformations, such a single affine expression is not expressive enough, but instead it is necessary to use different affine expressions for different subsets of the iteration space. Optimizations that require such piecewise schedules are called index set splitting [Griebl et al. 2000] transformations. Two recent works, hybrid-hexagonal tiling [Grosser et al. 2014] (see also Section 7.6) as well as work on time-tiling of periodic stencil computations [Bondhugula et al. 2014] both necessarily produce schedules with split index sets. To regenerate highly efficient control flow, both of these works benefit from the native support for piecewise schedules available in our AST generator. In the case of the work on periodic stencils, this control flow allowed the time tiling of swim, a large real-world application that is part of the SPEC 2000 benchmark suite. The use of a tiling scheme based on index set splitting has been reported to result in notable performance improvements that were not achievable using manual transformations because those were deemed to complicated and hard to debug.

```
#define mod(a, b) ((a) < 0 ? (a)+(b) : (a) >= (b) ? (a) - (b) : (a))

for (t = 0; t < N; t++)
  for (i = 0; i < 2N; i += 1)
S:  A[(t+1)%2][i] = A[t%2][mod(i+1, 2N)] + A[t%2][mod(i-1, 2N)]
```

Fig. 11: Original stencil code with wrapping dependences

Figure 11 shows a simple 1D stencil code similar to the loops in the swim kernel. The iteration domain of this kernel is

$$\{ S(t, i) : 0 \leq t < T \wedge 0 \leq i < 2N \}, \tag{61}$$

and the data dependences are

$$S(t, i) \rightarrow \begin{cases} S(t+1, i+1) & \text{if } i < 2N-1 \\ S(t+1, i+1-2N) & \text{if } i = 2N-1 \end{cases} \tag{62}$$

and

$$S(t, i) \rightarrow \begin{cases} S(t+1, i-1) & \text{if } i > 0 \\ S(t+1, i-1+2N) & \text{if } i = 0. \end{cases} \tag{63}$$

To now exploit reuse along the t dimension, it is necessary to tile the code across multiple iterations of t. Unfortunately, even though most data dependences have a short dependence distance ($(1, 1); (1, -1)$), there are also some with longer dependence distances ($(1, 1 - 2N); (1, -1 + 2N)$) at the iteration space boundaries. These latter dependences block the application of such a time tiling. To address this problem, the following piecewise schedule can be used:

$$S(t, i) \rightarrow \begin{cases} (t, i, 0) & \text{if } i < N \\ (t, 2N - i - 1, 1) & \text{if } i \geq N. \end{cases} \tag{64}$$

This schedule splits the iteration space into two parts and reverses the second part such that dependent iterations can be moved close to each other and the dependence distances are shortened to $(1, 0, 1); (1, 0, -1); (1, -1, 0); (1, 1, 0)$. We can directly use this schedule to generate the code in Figure 12 or, with now shortened dependences, use it as a base for further tiling.

With index set splitting natively supported by our AST generator, it is possible to generate such codes without being forced to introduce new virtual statements that model subsets of the iteration space to which different schedules have been assigned. Even though such preprocessing is possible, it breaks an abstraction barrier by requiring some AST generation steps to be taken in the scheduling optimizer. The use of virtual statements also hides from the AST generator the fact that different pieces of the schedule execute the same statement, which means there is no inherent reason to duplicate these statements. Instead, the AST generator can be instructed to avoid code duplication for rarely executed code paths (before or after parts when using isolation) and use code duplication to maximize the performance of the center part of a computation.

7.5. Run-Time Guards for Optimistic Transformations

When performing loop nest optimizations in the context of a regular C compiler, input programs often do not provide sufficient static information to verify the validity of interesting transformations. It is however still possibly to apply such transformations by optimistically assuming the required program properties and only executing the transformed code if the optimistic assumptions can be verified at run-time, falling back to

```
for (int c0 = 0; c0 < T; c0 += 1)
  for (int c1 = 0; c1 < N; c1 += 1) {
    S(c0, c1);
    S(c0, 2 * N - c1 - 1);
  }
```

Fig. 12: AST generated code after index set splitting (no tiling applied)

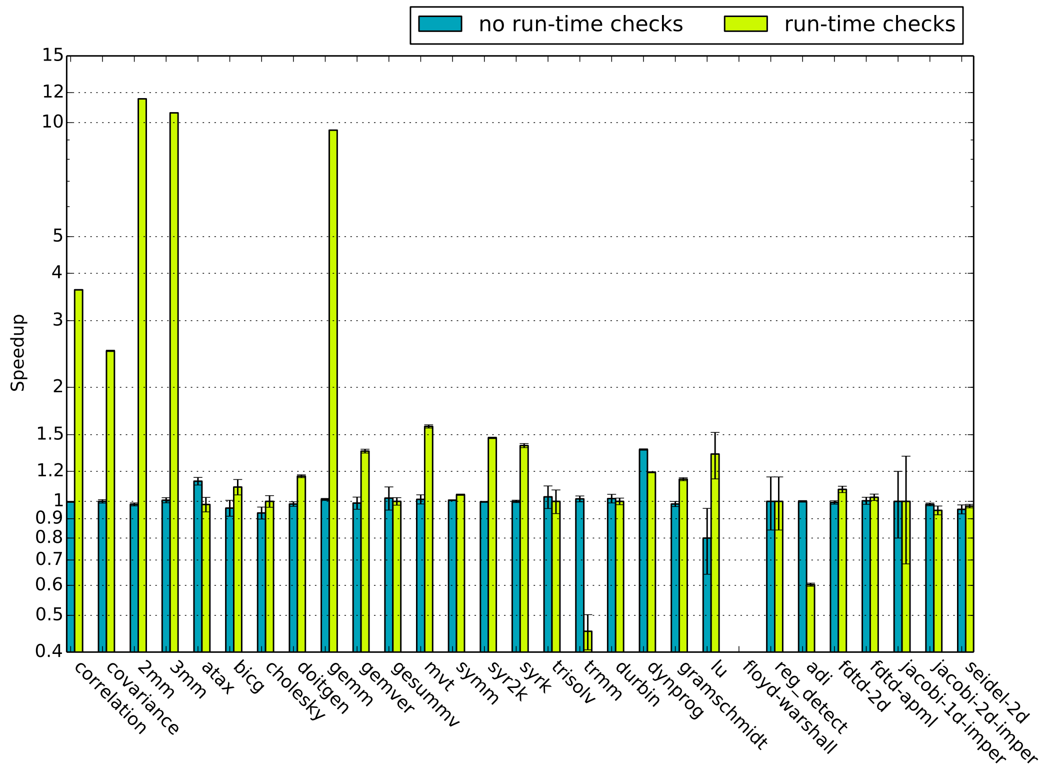

Fig. 13: Speedup of Polly with/without run-time checks in comparison to `clang -O3` on polybench 3.2

an unoptimized version of the code if this run-time verification fails. The AST generating facility of Section 5.10 can then be used to print efficient code for the collected conditions that guard the optimistically transformed code.

One illustration of this AST generation feature can be found in LLVM's Polly loop optimizer [Grosser et al. 2012]. Polly heavily depends on the AST generation of user-provided expressions which allows Polly to conveniently derive necessary run-time checks from constraints modeled as integer sets or from given affine expressions. Polly uses this facility to provide an "assumption tracking" framework where analyses and transformations performed in the context of Polly collect the assumptions they take. Polly then collects these assumptions in an integer set, leverages `isl` to simplify this set and finally uses the presented AST generator to derive AST expressions that verify the collected assumptions at run-time. Polly currently uses this framework to track assumptions about multi-dimensional arrays. It assumes that accesses into arrays of fixed size always remain within bounds. For polynomial access functions to one-dimensional arrays (e.g., `A[i + n * j]` to `A[]`), Polly aims to recover a multi-dimensional access shape and corresponding subscript expressions [Grosser et al. 2015], as this transformation often turns a polynomial dependence analysis problem into a linear problem which is in general easier to solve. Polly also leverages facilities to build AST expressions to deal with pointer aliasing, where it optimistically assumes the absence of aliases. To generate the run-time checks that prove absence of aliasing, Polly derives the set of array accesses in the loop program and computes for each array base pointer the lexicographically minimal and maximal (possibly multi-dimensional) subscript vector accessed during the execution of the loop program as a parametric quasi-affine expression, which are then translated to AST expressions using the presented AST generator.

To give the reader an idea of the additional optimizations enabled by the emission of run-time guards, we compile polybench 3.2 with Clang and Polly (r236223) once using plain `clang -O3`, then using clang and Polly without support for the generation of run-time checks and then again using clang and Polly with run-time check generation enabled. The resulting binaries are run single-threaded on an Intel Xeon E5430 CPU. We illustrate the results (median of 10 runs) in Figure 13. lu, jacobi-1d-imper and reg_detect show a very high variance due to the default run-time being very short and are consequently ignored in the subsequent analysis. Polly without run-time check generation enabled yields for most benchmarks the same performance as clang -O3, which is mostly due to Polly being unable to statically model the loop kernel and consequently not applying any transformation. When using run-time checks, 6 out of 29 benchmarks yield more than 50% speedup over clang -O3, and 3 additional benchmarks yield at least 20% speedup. The improvements in Polly here are caused by data-locality transformations (mostly tiling). The absolute speedup can vary significantly between platforms, depends mostly on the choice of the right schedule and good tile sizes and is specifically high as clang does not perform any loop tiling. There are also a couple of test cases where the run-time checks we emitted enabled further transformations, but where the transformations chosen by Polly's heuristics degraded performance compared to clang -O3 or to Polly without run-time check generation.

We emphasize that our contribution is not in finding the optimal code transformation or even the definition of an assumption tracking framework. However, the examples and numbers presented illustrate nicely what kind of optimizations can be *enabled* by our AST generator's support for the generation of AST expressions. There are different ways to derive run-time checks and there is not always a need to use a polyhedral AST generator to emit them. However, for the generation of more complex and (possibly partially redundant) conditions, the use of an integer set based framework to collect such conditions and the use of an AST generator to directly generate code for them has shown to work well in Polly. An AST generator based approach also makes future, more context-dependent optimizations directly available to the AST generation of arbitrary affine schedules, and also to the generation of user-provided conditions and expressions.

7.6. Performance Implications of AST Generation Strategies

To understand the performance implications of our new AST generation strategies, we analyze their impact on the run-time of generated code. We ensure a realistic scenario by analyzing a full end-to-end domain-specific compiler. As compiler we choose the stencil compiler introduced in Section 2. We remind the reader that this compiler is based on the general purpose compiler PPCG. To create code that is optimized for the domain of stencil computations, the computation of a generic execution schedule is replaced with the computation of a hybrid hexagonal/parallelogram execution sched-

		heat 2D		heat 3D	
AST generation options		GFLOPS	speedup	GFLOPS	speedup
a:	no options enabled	2.1	1.0x	5.5	1.0x
b:	all optimizations enabled	28.4	13.4x	21.0	3.8x
c_1:	all, except full/partial separation	21.5	10.2x	19.7	3.6x
c_2:	all, except IO unrolling	5.1	2.4x	10.5	1.9x
c_3:	all, except compute unrolling	15.9	7.5x	11.3	2.1x
c_4:	all, except modulo detection	29.1	13.7x	18.3	3.3x

Table III: AST generation strategy based performance (GFLOPS)

ule specifically optimized for the domain of stencil computations. Besides the domain specific schedule, the only other domain specific piece is the parametrization of our AST generator to isolate (Section 5.6) full tiles from partial tiles as well as to unroll (Section 5.5) compute and IO code. AST expression generation (Section 5.10) is used to specialize the access functions of statements, e.g., after unrolling or separation.

We perform the evaluation on an NVIDIA NVS 5200M GPU with CUDA compilation tools V5.5.0 using a heat 2D and a heat 3D stencil as benchmark and report results as the median over 10 samples. As performance results have shown large differences between two and three dimensional stencils, we choose two benchmarks to cover the most common dimensionalities. We limit ourselves to a single type of stencil, as the general tendency between different types of stencils does not vary enough to give additional insights for this analysis. For further performance results on different hardware and different stencil types, we refer to Grosser et al. [2014]. Table III shows the results of our analysis. We see in a that normal AST generation with no further specialization enabled yields very low performance with just 2.1 GFLOPS in the 2D case and 5.5 GFLOPS in the 3D case compared to b where we enable all optimizations and obtain 28.4 GFLOPS in the 2D case and 21.0 GFLOPS in the 3D case, a 13.4x speedup in the 2D case and a 3.8x speedup in the 3D case. To understand better where this speedup comes from we individually disable certain optimizations. In c_1 we disable full/partial tile separation, which reduces the performance by 24% for heat-2D and 6% for heat-3D. The larger change on 2D is due to the higher percentage of full tiles. In 3D, already a large amount of time is spent in partial tiles, so optimizations that speed up the execution of full tiles are less visible. In c_2 and c_3 we see that for heat 3D disabling either unrolling of IO or unrolling of compute reduces the performance by around 50%. For the 2D case, disabling unrolling of the compute code also reduces the performance by 44% and, even more importantly, without unrolling of the IO code over 80% of the performance is lost. This large performance difference is due to both the increased ILP after unrolling and the simplifications enabled by unrolling (see Figure 2). In c_4 we see that without modulo detection the performance for heat-3D is reduced by 13% and, surprisingly, slightly increased by 2% in 2D. The increase for heat-2D is due to register spilling caused by loop invariant code motion which again was made possible due to the simpler code after modulo detection. Allowing `nvcc`, the NVIDIA compiler, to use more registers prevents register spilling and modulo detection becomes again beneficial with a new peak performance of 29.4 GFLOPS for heat 2D. Overall, we see that just generating control flow using polyhedral scanning is by far not enough to generate high-performance GPU code. Instead, both polyhedral unrolling and specialization for full and partial tiles are highly important to obtain code of competitive performance. The fact that we achieve large speedups compared to an almost unoptimized code may not surprise the reader. However, we would like to note that such optimizations are currently not available in any other AST generator and even though some of them may be performed with the help of additional pre or post-processing, having all optimizations carefully interact inside the AST generator is beneficial. It enables a close integration of related optimizations, where for example the modulo detection has access to information about the currently unrolled loop. Furthermore, being able to gradually add optimizations without affecting the schedule of the program and, consequently, without affecting the correctness of the program, has shown useful when developing new AST generation based tools.

7.7. Generation Time

Although we have mostly focused on ease of use and quality of the generated AST, for completeness we also report some AST generation times. For this experiment, we take 64 of the test cases distributed with `CLooG` (those that can be handled by both `CLooG` 0.14.1 and `CodeGen+`) and sum the total AST generation time. For `CLooG` 0.14.1 (before our enhancements, using `PolyLib` as a backend), we obtain 0.3s using fixed size integer computations and 1.0s for arbitrary precision integers. For `CLooG` 0.18.1 (including some of our enhancements and using `isl` for set operations with arbitrary precision integers), we obtain 0.9s. For `CodeGen+`, we find 3.1s and for `isl`, 1.5s. We attribute the time difference with respect to `CLooG` to the fact that we have not yet implemented some of the heuristics of Vasilache et al. [2006] and that we are much more aggressive in our optimizations, resulting in better output code.

7.8. Generated Code Performance

It is difficult to compare the performance of code generated by different AST generators. First of all, the types of schedules we accept as input are much more generic than those supported by `CodeGen+` and old versions of `CLooG`. To be able to perform a comparison, we have therefore performed an experiment with relatively simple schedules. In particular, we have taken each of the PolyBench benchmarks [Pouchet 2012], applied the `isl` scheduler (a variation of the Pluto scheduler [Bondhugula et al. 2008]), tiled the outermost tilable loop and generated CPU code. For our AST generator, we set the atomic option on the tile loop band and the separate option on the innermost band. For `CodeGen+`, we use the default options, which is essentially to separate on the innermost loop. We also modified the source code of `CodeGen+` to produce nested calls to binary `min` and `max` macros instead of single calls to n-ary macros.

Another issue in comparing the performance of code generated by different AST generators is that the generated AST or source code still needs to be compiled by a compiler and the use of different compilers can lead to significantly different performance results. We therefore compiled the generated code using three different compilers, gcc 4.9.2 with options `-O3 -march=native`, clang 3.5 with options `-O3 -march=native`, and icc 13.1.0 with options `-fp-model strict -O3 -xHost`. The experiments were run on an AMD Opteron(tm) 6164 HE processor. For each benchmark and for each compiler, we used the standard PolyBench performance measurement, which is to run the experiment 5 times, remove the fastest and the slowest execution time and take the average of the remaining 3 execution times.

In an attempt to make the comparison as fair as possible, we applied the following tweaks. The context of the AST generation problem, i.e., the known constraints on the symbolic constants, is treated differently by `isl` and `CodeGen+`. While in our AST generator, the context is only used to simplify the generated AST, in `CodeGen+` the context constraints may end up in the AST. The default context, extracted by `pet` [Verdoolaege and Grosser 2012], contains bounds on the symbolic constants derived from their integer types. With this context, `CodeGen+` would use the lower bound on the symbolic constants in the lower bounds of the `for` loops, which moreover exposed a bug in `CodeGen+` that would replace the negative constants by positive constants, resulting in the loop

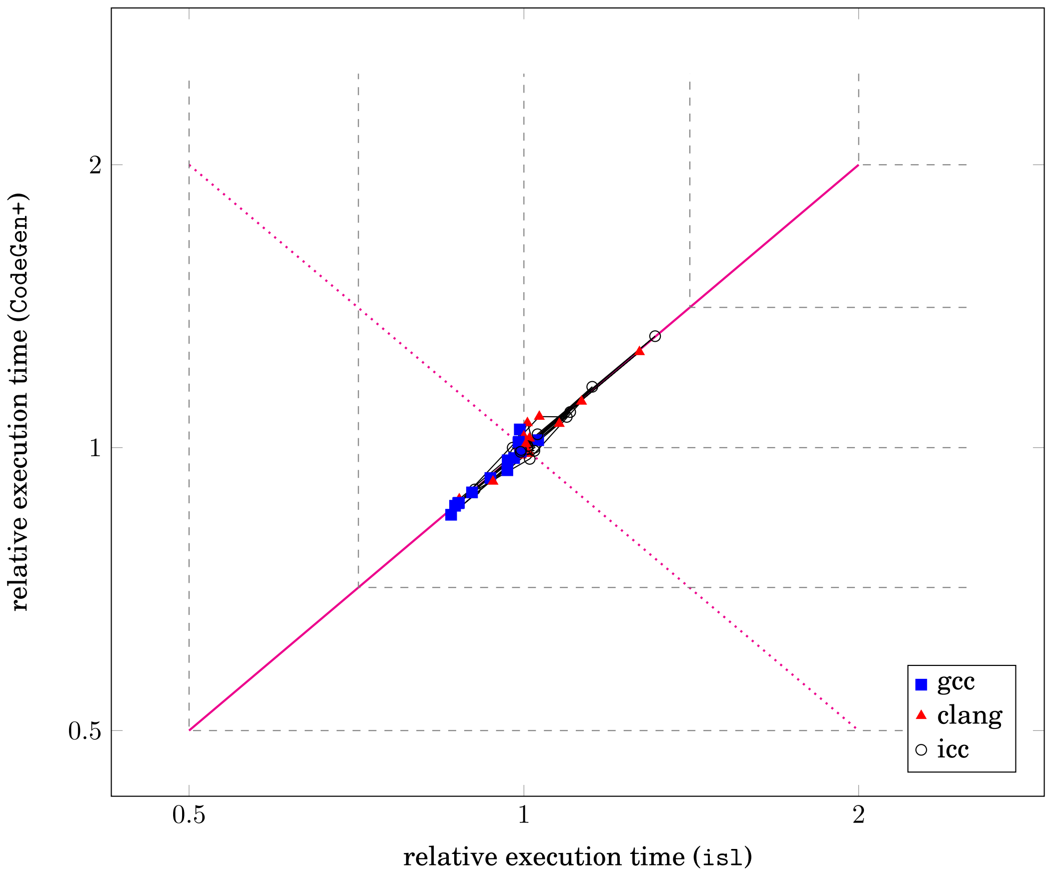

Fig. 14: Relative execution time – benchmarks with consistent performance

not being executed. Restricting to non-negative symbolic constants prevented this bug, but the upper bounds on the symbolic constants would still get used as upper bounds in the `for` loops resulting in execution times that were orders of magnitude longer. We therefore decided to drop the context from the `CodeGen+` input. On the other hand, we found that some compilers, especially gcc, are very sensitive to the way the `min` and `max` macros are implemented. In particular, on average gcc produces faster code when these macros only evaluate their arguments once, with extremes up to a factor of two. For some unknown reason, this effect is more noticeable on `isl`-generated code than on `CodeGen+`-generated code. In our experiments, we use macros that only evaluate their arguments once, using gcc extensions that are also supported by clang and icc.

The results are shown in Figure 14 and Figure 15. Figure 14 shows the performance results of the 16 benchmarks where the difference in performance between `isl` and `CodeGen+` is less than a factor of 1.1, whereas Figure 15 shows the results of the 14 benchmarks where this difference was larger for at least one compiler. The way the data is presented requires some explanation. Since we are mainly interested in the relative execution times, we have divided the execution times for each benchmark by the geometric mean of the 6 execution times (2 AST generators and 3 compilers). For each benchmark and each compiler, we plot a mark at the relative execution times and connect the three marks. By construction, the center of each resulting triangle

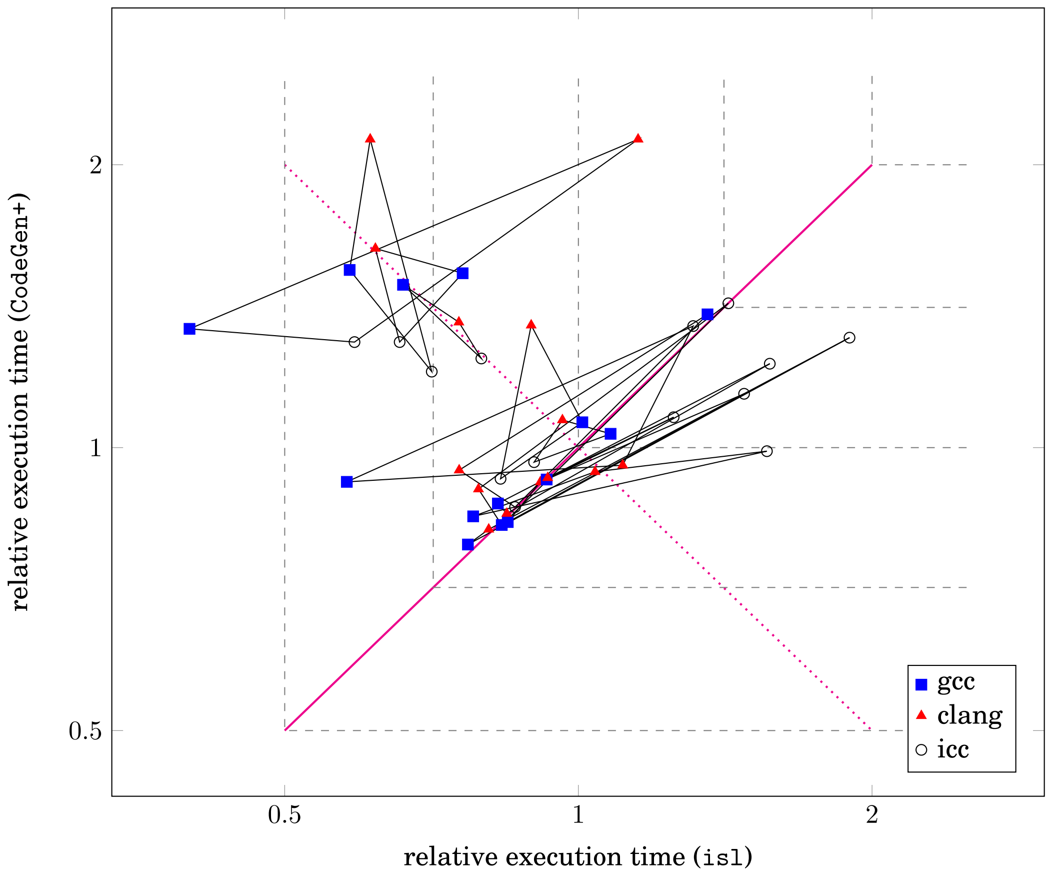

Fig. 15: Relative execution time – outliers

lies on the second diagonal (dotted magenta line). Points above the main diagonal (full magenta line) are those where `isl` generated code is faster. Points below the main diagonal are those where `CodeGen+` generated code is faster. The distance from the main diagonal is a measure for the performance improvement. Similarly, points that lie below the second diagonal are those where the corresponding compiler produces faster code than the average over the three compilers.

In general, we see that the performances of `isl` and `CodeGen+` generated code are very similar and that the differences between compilers is greater than the difference between AST generators. Only considering a single compiler could therefore lead to misleading results. For example, there are five benchmarks where icc produces significantly faster code on `CodeGen+` generated source code, but the other compilers produce much faster code for both sources. On the other hand, there is also one benchmark where icc produces significantly faster code on `isl` generated source code. There are also four benchmarks where the `isl` generated code performs better when compiled with all three compilers. A quick investigation reveals that this difference is due to `CodeGen+` failing to detect strides in some cases, whereas the stride detection of Section 5.3.1 is able to detect all strides.

While these figures may appear to suggest that `isl` generated code is faster than `CodeGen+` generated code, we would like to stress once more that these results depend on many factors and that we therefore do not want to make any such claims. For example, when using `min` and `max` macros that evaluate their arguments twice, the gcc compiled code is in general significantly slower. Moreover, this slow-down is more pronounced for `isl` generated source code, resulting in a couple of benchmarks showing the best performance (but worse than those in our figures) on `CodeGen+` generated source code.

We have also briefly investigated the experiments where Chen [2012] reports improved performance of `CodeGen+` generated code over `CLooG` (0.16.3) generated code. Our AST generator generates nearly identical code to `CodeGen+`'s, with the same performance. We did find that the code generation time was somewhat longer (8.7s vs 2.6s) on the `lu` case using their input formulation, where transformations have been applied on the iteration domains rather than in a schedule, including duplication of iteration domains to obtain the effect of unrolling. Using a more natural formulation in terms of a schedule tree, we can generate the same code in 1.4s.

8. RELATED WORK

There are two major approaches to generic AST generation, one that is based on a library for Presburger relations and that focuses on lifting control overhead up [Kelly et al. 1995; Chen 2012] and one that is based on rational polyhedra and that mainly tries to eliminate overhead top-down [Bastoul 2004; Quilleré et al. 2000]. Our approach can be seen as a combination, using the same separation algorithm of [Bastoul 2004; Quilleré et al. 2000], but built on top of a library for Presburger relations. Historically, we started by porting `CLooG` to `isl` and improving `CLooG`. Later, we built a new AST generator on top of `isl`. Both the original approaches only allow single disjunct contexts and schedules, with `CLooG` also not supporting existentially quantified variables. `CodeGen+` can handle such variables in certain cases, but as Chen [2012] does not discuss how such such variables are handled in general, the extent of support is unclear. In contrast, our AST generator supports the full generality of Presburger arithmetic, including existentially quantified variables and piecewise schedules.

In respect to the quality of the generated AST, some extensions proposed in the literature have not been implemented in `isl`. The components of Section 5.7 serve the same purpose as the "unisolate" procedure of Bastoul [2004, Section 4.2]. However, where the unisolate procedure tries to undo some separation, the components allow us to not even apply the separation. Moreover, no implementation of the unisolate procedure was ever made publicly available. Instead, recent versions of `CLooG` implement our components detection. Vasilache et al. [2006] propose several optimizations implemented in `CLooG` to reduce the AST generation time. Some of these optimizations are tailored to their encoding of schedule trees and are not needed when the schedule is represented as an explicit schedule tree. The same authors also propose an algorithm for removing modulo conditions, which on the one hand can be seen as a generalization of the shifted stride detection of Section 5.8, but on the other hand is based on a more restrictive polyhedral formulation. Zuo et al. [2013] describe several fine-tunings of `CLooG` tailored for high-level synthesis, only some of which are available in `isl`.

Kelly et al. [1995] and Chen [2012] as well as Quilleré et al. [2000] and Bastoul [2004] generate AST expressions as necessary to generate control flow for scanning the iteration space, but they do not expose any functionality to generate AST expressions for arbitrary user-provided piecewise quasi-affine expressions. We also are not aware of any work that uses the AST generation context to specialize AST expressions, in particular to optimize modulo operations and divisions as they appear in quasi-affine expressions. `CodeGen+` always generates expensive `intMod` calls and `CLooG` only introduces a `%` operator in cases where the result of the operator is compared to zero.

Polyhedral unrolling in an AST generator has been proposed (without software being made available) by Vasilache et al. [2006] for the special case of a unimodular schedule where a dimension that has a single lower and single upper bound offset by a constant non-parametric distance can be fully unrolled. In our work we presented polyhedral unrolling for schedules defined by arbitrary Presburger maps, with support for unrolling in the presence of multiple lower bounds, unrolling in the presence of strides and unrolling `for` loops with a bounded, non-constant number of iterations using conditional statements. User-directed isolation of arbitrary subsets of the iteration space as such has not been implemented in polyhedral AST generators. The automatic separation used by Bastoul [2004] regularly introduces specialized code versions, but the user can only control the amount of separation and not the subsets that are separated from each other. Full/partial tiling has been discussed as an independent transformation by Ancourt and Irigoin [1991] as well as Goumas et al. [2003] and, combined with unrolling, by Jiménez et al. [2002]. In the context of parametric tiling [Kim et al. 2007; Renganarayanan et al. 2007; Hartono et al. 2010], full/partial tile separation has been researched in AST generators specialized for this use case. We are not aware of any work that uses a generic isolation feature provided by a polyhedral AST generator to perform full/partial tile separation. As parametric tiling techniques commonly rely on polyhedral AST generators, the same isolation techniques may be useful in the context of parametric tiling.

We are not aware of any work that provides configurability on such a fine grained level. CLooG [Bastoul 2004] originally allowed per-dimension level control over separation and recently gained per-statement control. Chen [2012] allows per loop level control over the amount of control flow. Different AST generation strategies for different subtrees of the generated AST are to our knowledge unique to our work. Also, giving the user the ability to enforce an “atomic” AST generation strategy to minimize code size or to enforce unrolling is new.

9. CONCLUSION

This work significantly widens the scope of polyhedral AST generation. It does so by extending traditional control flow generation to the full generality of Presburger arithmetic. In particular, we provide support for piecewise affine schedules as well as schedules with complex uses of existentially quantified variables, opening AST generation to new application areas and more sophisticated program optimizations, and enhancing its reliability—the ability to predictably generate highly efficient control flow. Our work also improves the quality of the generated imperative code by presenting optimizations for shifted strides and components. We also acknowledge that optimization problems are not limited to control flow restructuring, but also require changes to data access functions: to support such optimizations, we propose facilities to generate efficient AST expressions from piecewise quasi-affine forms. Finally, we improve on the state-of-the-art techniques for recovering divisions and modulo expressions in the generated code, and apply these to the optimization of index expression that commonly appear in the context of explicitly managed caches. Overall, we widened the scope of generic AST generators.

However, to implement domain or target specific optimizations that reach peak performance, it is often necessary to heavily specialize the generated code. For this we allowed the AST generator to be parametrized to perform loop unrolling and partial evaluation of loop iterators in a very general, polyhedral setting. Furthermore, we presented how to separate certain parts of the code and showed how to use this separation to generate specialized code for full and partial tiles. By allowing the specialization of user-provided AST expressions according to the context they are generated in, the same feature can also be used to generate specialized code for boundary conditions. As maximal specialization may not always be best, we make AST generation choices such as separation, unrolling and also atomic execution configurable on a fine-grain level. Each individual contribution is by itself useful, but only the integration in a single AST generator ensures their seamless interaction. As demonstrated on hybrid hexagonal/classical tiling, the resulting AST generator can implement complex domain-specific optimizations, and is a powerful alternative to the development of a problem-specific code generator.

Acknowledgements

This work was partly funded by a gift received by LIACS from Intel Corporation, by a Google Europe Fellowship in Efficient Computing, by the European FP7 project CARP id. 287767 and the ARTEMIS project COPCAMS id. 332913 as well as by PASC, the Swiss Platform for Advanced Computing.

REFERENCES

Corinne Ancourt and François Irigoin. 1991. Scanning Polyhedra with DO Loop. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '91)*. 39–50.

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 40.

Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 7–16.

Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. 2014. Tiling and Optimizing Time-iterated Computations on Periodic Domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 39–50.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelization and Locality Optimization System. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113.

Chun Chen. 2012. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, 499–508.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *A framework for composing high-level loop transformations*. Technical Report 08-897. University of Southern California.

Alain Darte, Yves Robert, and Frédéric Vivien. 2001. Loop Parallelization Algorithms. In *Compiler Optimizations for Scalable Parallel Systems*, Santosh Pande and Dharma P. Agrawal (Eds.). Springer Verlag, New York, NY, USA, 141–171.

Paul Feautrier. 1988. Parametric Integer Programming. *RAIRO Recherche Opérationnelle* 22, 3 (1988), 243–268.

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

Paul Feautrier and Christian Lengauer. 2011. The Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1581–1592.

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. *International Journal of Parallel Programming* 34, 3 (June 2006), 261–317.

Georgios Goumas, Maria Athanasaki, and Nectarios Koziris. 2003. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems* 14, 10 (2003), 1021–1034.

Martin Griebl, Paul Feautrier, and Christian Lengauer. 2000. Index set splitting. *International Journal of Parallel Programming* 28, 6 (2000), 607–631.

Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, 66:66–66:75.

Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 28.

Tobias Grosser, Louis-Noël Pouchet, J. Ramanujam, P Sadayappan, and Sebastian Pop. 2015. Optimistic Delinearization of Parametrically Sized Arrays. In *29th International Conference on Supercomputing (ICS 2015)*.

Albert Hartono, Muthu Manikandan Baskaran, J Ramanujam, and Ponnuswamy Sadayappan. 2010. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *Proceedings 16th International Parallel and Distributed Processing Symposium (IPDPS '10)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12.

Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th international ACM conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 13–24.

Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 311–320.

ISO ISO. 1999. IEC 9899: 1999: Programming languages C. *International Organization for Standardization* (1999).

Marta Jiménez, José M Llabería, and Agustín Fernández. 2002. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 4 (2002), 409–453.

Wayne Kelly and William Pugh. 1995. A unifying framework for iteration reordering transformations. In *IEEE First International Conference on Algorithms and Architectures for Parallel Processing (ICAPP '95)*, Vol. 1.

W. Kelly, W. Pugh, and E. Rosser. 1995. Code Generation for Multiple Mappings. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*. IEEE Computer Society, Washington, DC, USA, 332–341.

DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-level Tiling: M for the Price of One. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 51, 12 pages.

Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 127–138.

Vincent Loechner and Doran K Wilde. 1997. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming* 25, 6 (1997), 525–549.

L.-N. Pouchet. 2012. PolyBench/C 3.2. (2012). http://www.cs.ucla.edu/~pouchet/software/polybench/.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM Press, Tucson, Arizona, 90–100.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society press, San Jose, California, 144–156.

William Pugh and Evan Rosser. 1997. Iteration Space Slicing and Its Application to Communication Optimization. In *Proceedings of the 11th International Conference on Supercomputing (ICS '97)*. ACM, New York, NY, USA, 221–228.

William Pugh and David Wonnacott. 1994. Static analysis of upper and lower bounds on dependences and parallelism. *Transactions on Programming Languages and Systems (TOPLAS)* 16, 4 (1994), 1248–1278.

Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of Efficient Nested Loops from Polyhedra. *Int. Journal of Parallel Programming* 28, 5 (Oct. 2000), 469–498.

Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Parameterized tiled loops for free. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 405–414.

Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. 2014. Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 287–298.

Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral Code Generation in the Real World. In *Proceedings of the 15th International Conference on Compiler Construction (CC '06)*, Vol. 3923. Springer, Vienna, 185–201.

Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Strout. 2014. Non-affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, 185:185–185:194.

Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010 (Lecture Notes in Computer Science)*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.), Vol. 6327. Springer, 299–302.

Sven Verdoolaege. 2011. Counting Affine Calculator and Applications. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France.

Sven Verdoolaege. 2015. Integer Set Coalescing. In *Proceedings of the Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*. Amsterdam, the Netherlands.

Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Proceedings of the Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France.

Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule Trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria.

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. 2012. Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 3, Article 11 (Nov. 2012), 35 pages.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (Jan. 2013), 54:1–54:23.

David Wonnacott. 2002. Achieving Scalable Locality with Time Skewing. *International Journal of Parallel Programming* 30, 3 (2002), 181–221.

Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*.

Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. 2013. Improving Polyhedral Code Generation for High-Level Synthesis. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13)*. IEEE Press, Piscataway, NJ, USA, 15:1–15:10.