Schedule Trees

Conference Paper · January 2014

DOI: 10.13140/RG.2.1.4475.6001

4 authors, including:

Sven Verdoolaege
National Institute for Research in Computer Science and Control
63 PUBLICATIONS **940** CITATIONS

SEE PROFILE

Schedule Trees

Sven Verdoolaege
INRIA, École Normale Supérieure and
KU Leuven
sven.verdoolaege@inria.fr

Serge Guelton
École Normale Supérieure and
Télécom Bretagne
serge.guelton@telecom-bretagne.eu

Tobias Grosser
INRIA and École Normale Supérieure
tobias.grosser@inria.fr

Albert Cohen
INRIA and École Normale Supérieure
albert.cohen@inria.fr

ABSTRACT

Schedules in the polyhedral model, both those that represent the original execution order and those produced by scheduling algorithms, naturally have the form of a tree. Generic schedule representations proposed in the literature *encode* this tree structure such that it is only implicitly available. Following the internal representation of `isl`, we propose to represent schedules as *explicit* trees and further extend the concept by introducing different kinds of nodes. We compare our schedule trees to other representations in detail and illustrate how they have been successfully used to simplify the implementation of a non-trivial polyhedral compiler.

1. INTRODUCTION

The polyhedral model [10] 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 [16] or Presburger relations [19]. 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 a relative execution order to the statement instances.

The shape and representation of a schedule varies with the way in which the schedule is computed. However, most scheduling algorithms recursively decompose a dependence graph, at each level separating the graph into (strongly connected) components and computing a partial schedule for each component separately. This partial schedule is usually an affine function (possibly quasi-affine and/or piece-wise). The complete schedule is then obtained as some form of concatenation of the partial schedules, mapping statement instances to a *schedule space*. The order of two statement instances in the complete schedule is determined by the outer partial schedule that yields a different value for the two instances or the outer pair of components that separates the two instances, whichever appears outermost. An overview of several early such algorithms is provided by Darte et al. [7]. When constructing tilable bands (e.g., the Pluto algorithm [3]), the partial schedules are multi-dimensional affine functions and the order determined by a partial schedule is given by the lexicographic order on its function values.

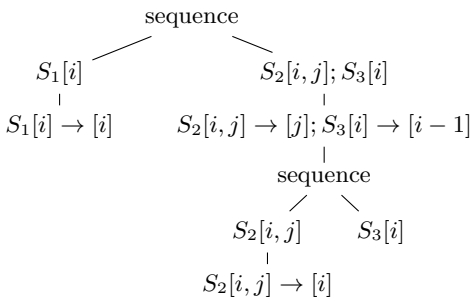

Figure 1: Example schedule tree representation

The above description suggests a representation of a schedule in the form of a *tree*, with each node representing a partial schedule and the order of the components determined by the order of the children of a node. Such a representation also seems the most natural way to represent the original order of a program, when extracted from some form of abstract syntax tree (AST). We are however unaware of any prior work that explicitly operates on such schedule trees (apart from our own "band forests" in `isl`). Instead, the schedule trees are *encoded* in one way or another and all operations are performed on the *encodings* of the schedule trees. Depending on the chosen encoding, these operations quickly become cumbersome and/or hard to understand.

In this paper, we propose to use an *explicit* representation of a schedule as a tree and to perform all operations directly on this tree. We argue that such a representation is *more natural*, *more practical* and *easier to understand*. Figure 1 illustrates a schedule tree representation of the schedule

$$\begin{array}{lllll} T_1 : \{[i] & \to [0, i & & &]\} \\ T_2 : \{[i,j] & \to [1, j, & 0, i &]\} \\ T_3 : \{[i] & \to [1, i-1, & 1 &]\} \end{array}$$

IMPACT 2014
Fourth International Workshop on Polyhedral Compilation Techniques
Jan 20, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

`http://impact.gforge.inria.fr/impact2014`

in Kelly's abstraction [14, 15] or

$$\Theta^{S_1} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \Theta^{S_2} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \Theta^{S_3} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$$

in the representation of Girbal et al. [11]. We first describe the general concept of a schedule tree and show how it generalizes the different schedule representations that have been proposed in the past. We subsequently propose a specific *instance* of the schedule tree concept that will be used in future versions of `isl` [22]. This representation is *more general* than earlier proposals, allowing an explicit representation of subtrees that can be executed in parallel and the introduction of additional symbolic constants in subtrees. Finally, we show how we used this new representation to *simplify* the implementation of `PPCG` [25], significantly improving the maintainability of the tool and enabling future extensions.

2. SCHEDULE USES

Depending on the framework used, there may be a single iteration domain containing all statement instances or there may be an iteration domain per statement. Invariably, though, the elements of an iteration domain are (possibly named) vectors of integers, called *iteration vectors*. In some approaches, these elements determine an implicit execution order, but this means that whenever a transformation is applied that changes the execution order, the iteration domains themselves and everything that depends on the iteration domains (such as access relations and dependences) need to be updated. In this paper, we will therefore assume that, as is common practice, the execution order is determined by an explicit schedule that maps the elements of the iteration domains to some other objects with an implicit execution order. Note that a schedule only prescribes a *relative* execution order and that there are therefore typically an infinite number of ways to express the same execution order, independently of the chosen schedule representation. In this section, we describe some uses of schedules in general that we will use to compare the different schedule representations in Section 3.

2.1 Original Execution Order

Although much work has been devoted to automatic scheduling techniques that construct a schedule directly from the dependence graph, the ability to interactively perform polyhedral transformations starting from the original execution order is useful for teaching or manual exploration. Some implementations of dataflow analysis [8] (e.g., that of `isl`) also start from a polyhedral representation involving some form of schedule (even though it may be more advisable to perform the dataflow analysis before or during the extraction of a polyhedral representation from an AST using techniques similar to lazy array data-flow analysis [17] or array region analysis [6]). Our schedules therefore need to be able to represent the original execution order.

When extracting a polyhedral model from an imperative program, we mainly need to deal with two constructs in terms of execution order, compound statements and loops. In order to model the effect of compound statements, a schedule needs to be able to express a sequence, i.e., that one set of statement instances should be executed after some other set of statement instances. To model the effect of a loop, the schedule needs to be able to express an order defined by an affine function on the iteration vectors. In the simplest case, the iteration vectors are composed of the values of the iterators of the enclosing loops. In this case, the affine function simply projects the iteration vector onto the iterator of the loop that needs to be modeled. In general, an iteration vector can be any sequence of integers that uniquely identifies a statement instance and a more complicated function may be required to express the effect of a loop.

2.2 Transformations

A polyhedral representation can be transformed either by constructing a new schedule or by modifying some previously obtained schedule. This other schedule may be a schedule representing the original execution order or it may be the result of an earlier transformation.

2.2.1 Schedule Construction

As an example of an automatic scheduling algorithm, let us sketch a general overview of the "Pluto algorithm" [3]. The algorithm takes a dependence graph as input and recursively constructs schedule *bands*. The dependence graph expresses which statement instances depend on which other statement instances and therefore need to be executed after those other statement instances. The nodes of the dependence graph are composed of the statements, while the edges carry the dependence relations.

At each level of the recursion, the algorithm first checks for (weakly connected) components in the dependence graph. These components do not depend on each other in any way and can therefore be scheduled independently. Within each component, the algorithm looks for strongly connected components (SCCs) and (optionally) marks them to be executed in their topological order. For each (group of) SCC(s), the algorithm then constructs a sequence (or band) of one-dimensional affine functions such that each of these functions respects the dependences independently of the other functions in the same band, they are linearly independent of each of the other functions in the same and in outer bands, and such that they optimize some optimization criterion. After the construction of a band is completed, the dependence graph is updated to only contain dependences between pairs of statement instances that are mapped to the same function values by the current band and the process repeats.

The constructed schedule is therefore (at least conceptually) a tree that recursively consists of collections of statement instances that can be executed in any order, sequences of statement instances that need to be executed in the specified order and multi-dimensional affine functions.

2.2.2 Schedule Modification

Given a schedule (either corresponding to the original execution order or constructed from a dependence graph), we may want to apply a series of additional transformations. In keeping in line with a clear separation between the statement instances (in the iteration domain) and the order in which they are executed (defined by the schedule), these transformations need to be expressed as transformations on the schedule itself.

The transformations that we may want to apply include affine (typically uniform) transformations, statement reorder-

ing, fusion, distribution, index set splitting, strip-mining and tiling. Most of these transformations do not require any additional constructs beyond collection, sequence and multi-dimensional affine functions. The only exceptions are strip-mining and tiling. These transformations require the use of integer divisions and/or modulo operations and therefore require (explicit or implicit) *quasi-affine* expressions. Note that we only consider non-parametric strip-mining and tiling here.

2.3 AST Generation

AST generation takes an iteration domain and a schedule as input and produces an AST that visits each element of the iteration domain in the order specified by the schedule. This operation is also known as polyhedral scanning [1, 4] or code generation [2]. The constraints on the schedule representation imposed by AST generation are not so much in what the schedule should be able to express, but in the kind of constructs for which an AST can be generated. Clearly, generating an AST for a collection of statement instance groups or for a sequence of such groups is trivial. Piecewise quasi-affine schedule functions can also be handled by standard AST generators [2, 4].

The iteration domain and the schedule may refer to symbolic constants (also known as parameters). If the iteration domain is non-empty for only some values of these symbolic constants, then the generated AST may contain explicit conditions on the symbolic constants. Most AST generators allow the user to avoid the generation of such conditions by providing the AST generator with known constraints on the symbolic constants. This additional piece of information is known as the *context* and is usually passed separately to the AST generator. A final piece of information required by the AST generator is a set of options that control the way the AST is generated.

3. SCHEDULE REPRESENTATIONS

Many different schedule representations have been proposed in different contexts. Some of these proposed representations only serve as the input or output to a given algorithm and are therefore typically unsuitable as a generic schedule representation. Other proposals, such as "Kelly's abstraction" [14,15], "$2d+1$-schedules" [11] and "union maps" [23], have been specifically designed as generic representations. In this section, we compare some of these representations.

3.1 Properties

In particular, we compare the following aspects of the schedule representation

Scatteredness While some representations consist of a single schedule object, in other representation the schedule information is spread over different objects, typically one object for each statement.

Compositionality Compositionality is usually interpreted to mean that the same schedule representation can be used as both the input and the output of schedule transformations. In some cases, the schedule transformations themselves can be composed before being applied to the schedule.

Partial schedules Some schedule representations are very restrictive and only allow a limited set of predefined, implicit partial schedules. Other representations allow affine, quasi-affine or even piecewise quasi-affine partial schedules. Recall that quasi-affine schedules are required to express strip-mining or tiling. Some representations also restrict the partial schedules to a single dimension.

Sequence Many schedule representations do not support an explicit representation of sequence. Instead, each group of statement instances in the sequence is assigned a distinct increasing number. These increasing numbers may be assigned as (part of) a regular partial schedule, or they may be specified through a dedicated mechanism, typically only allowing a single constant for all instances of a statement. That is, these dedicated mechanisms typically do not allow the set of instances of a given statement to be broken up into two or more parts.

Collection Very few schedule representations are able to express that groups of statement instances can be executed in arbitrary order with respect to the other groups. Instead, such collections are usually encoded in the same way as sequences, fixing a particular execution order of the groups.

Injectivity Some early schedule representations explicitly allowed different statement instances to be assigned the same value in order to express inner parallelism. Other representations treat inner parallelism in the same way as other forms of parallelism and expect the statement instances that can be executed in parallel (at a given position in the schedule) to be assigned different values using a partial schedule, but somehow mark one or more dimensions in this partial schedule as parallel.

Singlevaluedness Some schedule representations allow a given statement instance to be assigned more than one value by the schedule, i.e., they allow the statement instance to be executed more than once.

Lexicographic order Two schedule values (either within a partial schedule or over the entire schedule) are usually compared based on the lexicographic order. In some representations, this lexicographic order is only defined on vectors of the same dimension. That is, the (partial) schedule is expected to have the same dimension across all statements. This corresponds to the standard definition of lexicographic order as found in most textbooks. We will call this a strict interpretation of lexicographic order. In other representations, the schedule vectors can have different dimensions and then the shortest vector is compared to the prefix of the longest vector of the same size as the shortest vector. We will call this a relaxed interpretation of lexicographic order.

3.2 Comparison

Some of the earliest schedule representations within the context of polyhedral compilation appear in Feautrier's work [9]. The input schedule is mostly implicitly encoded in the iteration domains, augmented with the number of shared loops for each pair of statements and the textual order of

	Kelly	$2d+1$	union map	band forest	schedule tree
scatteredness	per statement	per statement	single object	single object	single object
compositional	schedule	schedule	transformation	schedule	schedule
partial schedule representation	p.w.q.a.	affine	p.w.q.a.	p.w.q.a.	p.w.q.a.
partial schedule dimension	arbitrary	1	arbitrary	arbitrary	arbitrary
sequence	cst per statement	cst per statement	p.w.q.a.	p.w.q.a.	explicit
collection	n/a	n/a	n/a	implicit	explicit
injective	yes	yes	yes	yes	yes
single-valued	yes	yes	no	yes	mostly
total	possibly	yes	no	yes	internally
lexicographic order	relaxed	relaxed	strict	relaxed	relaxed

Table 1: Comparison of some generic schedule representations

each pair of statements. The relative order of two statement instances is determined by the lexicographic order on the prefixes of their iteration vectors of length equal to the number of shared loops and by the textual order of the statements. The output schedule is a multi-dimensional piecewise quasi-affine schedule, with relaxed lexicographic order, implicit inner parallelism and sequence encoded as any other schedule row.

The first compositional representation appears to have been proposed by Kelly [14,15]. Each statement keeps track of its part of the schedule. The partial schedules may be any multi-dimensional piecewise quasi-affine functions. Sequence is encoded by special schedule dimensions that are marked as "syntactical" and that assign the same constant value to all instances of a statement. Schedule values are compared using a relaxed lexicographic order. Although an explicit index set splitting operation is provided, index set splitting typically happens implicitly through the use of a piecewise partial schedule. When transforming a subtree of the schedule, the subtree is identified by the statements that are transformed by that subtree. The schedules appear to be padded with zeros prior to being sent to the AST generator.

The "$2d+1$" representation [11] can be seen as a further specialization of Kelly's abstraction. In particular, the partial schedules are restricted to one-dimensional purely affine functions (compared to the multi-dimensional piecewise quasi-affine functions of Kelly). The single-dimensional partial schedules are interleaved with constant statement level dimensions that express sequence. The restriction to purely affine functions means that they are unable to express strip-mining and tiling in the schedule itself and instead have to resort to a modification of the iteration domains, undermining the separation between iteration domains and schedule and the compositionality of their approach. The restriction to one-dimensional partial schedules (between statement level dimensions) means that unimodular transformations involving more than one loop dimension need to be applied across statement level dimensions. When transforming a subtree of the schedule, the subtree is identified by the values of the outer shared statement level dimensions (the "β-prefix"). Although this may appear to be more generic than using the statements involved, this is in fact not the case as each statement can only have a single β-prefix.

The "union map" representation [23] essentially pads the per-statement schedules of Kelly with zeros to ensure that all schedules have the same dimension and then combines the per-statement schedules into a single schedule object. This

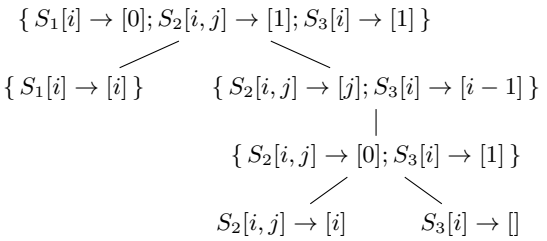

Figure 2: Band forest representation of the schedule in Figure 1

single object is a binary relation on tuples that maps named integer vectors (with the names representing the statements), to integer vectors of a fixed length. The main advantage of this representation is that it is not tailored to schedules. In particular, the same abstraction is also used to represent access relations and dependence relations, allowing for a uniform manipulation. Moreover, the *changes* to the schedule are also represented using the same abstraction and can be combined prior to being applied to the schedule. The non-specificity to schedules is also the main disadvantage of the union map representation. Whereas the tree structure is still visible in Kelly's abstraction and in the $2d+1$ representation through the marking as syntactical dimensions and the implicit statement level dimensions, this structure is completely hidden in the union map representation. The mapping to a single schedule space also causes local transformations to potentially have a global effect. For example, if some part of the schedule tree is tiled, increasing the total number of schedule dimensions, then the other parts of the schedule need to be padded to maintain a single schedule space.

The "band forest" abstraction that was available in versions 0.07 to 0.12 of `isl` builds on top of the union maps, but makes the tree structure explicit. It was used to represent the schedules computed by `isl`'s scheduler, which is very similar to the Pluto scheduler. Each node in the tree represents a tilable band, with a partial schedule represented by a union map. Siblings (including the roots of the forest) represent groups of statement instances that can be executed in parallel. Sequence is still encoded, in particular as a single-dimensional band with a union map that assigns constant values to groups of statement instances. The children of such a band can be executed in parallel because the

band itself takes care of the ordering. The only operations that were made available for this abstraction are splitting a band into two nested bands and tiling a band. Figure 2 shows an example of a band forest.

Table 1 provides an overview of the above comparison, including a comparison to the schedule trees of Section 4. Note that while union map schedules are typically single-valued, this is not strictly required. While the band forest is essentially a tree of union maps and it is technically possible for the band forest not to be single-valued, the band forests that are constructed in practice are always single-valued. The "total" row indicates whether the schedule is a total function. When using union maps, it is customary to encode the iteration domain in the domain of the schedule so that no separate iteration domain object is required. It is then also possible to select a subset of some larger external iteration domain. This is also possible in the case of Kelly's abstraction, but it is not clear if this is the intended use. The abstractions with a relaxed lexicographic order use a strict order within the partial schedules.

CLooG [2] and the union maps based `isl` code generator impose a strict lexicographic order on its input, requiring users to pad the schedules. Omega's code generation [4] does not impose a strict order. URUK [11] uses "$2d+1$" schedules, with subtrees identified by prefixes. Pluto [3] allows partial schedules (i.e., bands) of any dimension, but maintains the restriction of purely affine partial schedules, therefore also requiring a modification of the iteration domains to express tiling. A strict lexicographic order is imposed and subtrees are identified by sets of statements and a loop depth. PoCC [18] appears to be using the schedule representation of older versions of Pluto where bands were defined globally across the entire schedule tree. It also has some support for "reentrancy" [21], i.e., a conversion back and forth between polyhedral representation and AST. The Graphite [20] internal representation is similar to Kelly's, with the "syntactic" label replaced by a "loop nest tree" (only for the original schedule). The CHiLL [5] representation is also similar to Kelly's, except that partial schedules are single-dimensional as in the $2d+1$-schedules, with unimodular transformations applied across partial schedules. Subtrees are identified by sets of statements and a loop depth. Polly [13] essentially uses union maps, but breaks them up over the statements. Incremental transformations in AlphaZ [26] are performed through a modification of the iteration domains. It is also possible to specify an additional purely affine schedule function with strict lexicographic order.

4. SCHEDULE TREE REPRESENTATION

Based on our experience with the different kinds of schedule representations, we designed a new representation that, like the band forest, maintains an explicit tree structure. Unlike the band forest, however, the schedule tree has different types of nodes that allow for an easier manipulation and the ability to attach more information to the tree. In particular, sequence and collection are represented explicitly as nodes rather than being encoded in a band or implied by the tree structure. Our description is still somewhat preliminary, but our use of schedule trees in `PPCG` provides some initial evidence of the usefulness of this representation.

4.1 Nodes

The following types of nodes are available in the new schedule trees.

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. The context node typically appears as the root of the schedule tree, but as we will see in Section 5, it can also be useful to introduce additional context nodes in the tree. As a convenience to the user, the outer context node may also be left out, in which case it is assumed that the symbolic constants used in the tree can take on any value.

Domain A domain node introduces the statement instances that are scheduled by the descendants of the domain node.

Filter A filter node selects a subset of the statement instances introduced by outer domain 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. As we will see in Section 5, it can in some cases also be useful to introduce other filters in the tree.

Sequence A sequence node expresses that the children of the node should be executed in order. These children must be filter nodes, with typically 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 nodes and retained by outer filter nodes. This partial schedule may be piecewise quasi-affine, but is total on those statement instances. Additionally, a band node contains properties of the band and options that control the AST generation. The set of properties includes whether the band is tilable and which of the band dimensions may be executed in parallel. The AST generation options mainly control whether a band dimension should be separated or whether it should be unrolled.

Mark A mark node allows the user to mark specific subtrees of the schedule tree.

Sequence and set nodes have one or more children. The other types of nodes have at most one child.

The inclusion of context, domain and AST generation options in the schedule tree means that only a single object needs to be passed to the AST generator. This is especially important for the options. The original interface to the `isl` AST generator allowed for a very generic specification of options based on constraints on the schedule dimensions. The only purpose of this generic mechanism, however, was to be able to express that some options should be applied to a specific node of the schedule tree encoded in the union map. By attaching the options directly to the band nodes, the complexity of the generic option mechanism (mostly for the user, but also for the implementation), can be avoided completely. In particular, if the schedule is modified after some options have been set, then there is no longer any need to try and apply the same transformation to the options description as the local options automatically remain attached to the correct band node.

Note that while the schedule tree now contains both domain information (in the domain nodes) and schedule information (in the band nodes), the information is still kept in separate nodes so that, internally, the schedule is a total function on the domain. The explicit representation of a sequence means that the scheduler does not need to encode the sequence as a piecewise constant partial schedule only to have the AST generator identify this piecewise constant partial schedule as a sequence. Note that this type of node is only meant as a convenience for cases where the scheduler or user wants to impose an explicit sequence. If an automatic scheduler constructs a partial piecewise affine schedule in one of its substeps that just happens to be piecewise constant, then the corresponding band node does not have to be converted to a sequence node. The explicit representation of a collection rather than forcing an arbitrary order allows the user and the AST generator to reorder the children, without having to re-analyze the dependences.

The basic schedule tree representation (without the extensions of Section 5) has the same expressivity as Kelly's abstraction, union maps and band forests. The main advantages are the ease of manipulation and the potential for extensions. Note that some of the extensions of Section 5 may limit the kinds of operations that can be applied to the tree. These operations then need to be applied to the tree before such extended nodes are introduced.

4.2 Operations

The following operations are available to modify schedule trees. It is important to note that none of these operations modify the domain node(s). That is, even though the domain information is integrated in the schedule object, it is still kept separate from the actual scheduling information.

- Insert a context, domain, filter or mark node at a given position.
- Apply a piecewise quasi-affine transformation to a band node. The input space of the transformation is equal to the schedule space of the original partial schedule. The output space of the transformation becomes the new schedule space. This operation can be used to implement any unimodular transformation on the band, but also strip-mining and tiling within the band, possibly combined with index set splitting.
- Split a band node into two nested band nodes, each node holding a part of the original output domain.
- Combine two nested band nodes into a single band node. This is the opposite of a Split.
- Tile a band node. This operation is essentially the same as tiling the band within the node through the application of a piecewise quasi-affine transformation and then splitting the node into a band corresponding to the tile loops and a band corresponding to the point loops.
- Fuse two bands. This operation pushes an explicit order on a pair of bands down, combining the two bands into a single band. This operation typically has the effect of loop fusion on AST generation. The bands need to be grandchildren of the same sequence with adjacent (filter) parents or grandchildren of the same set.

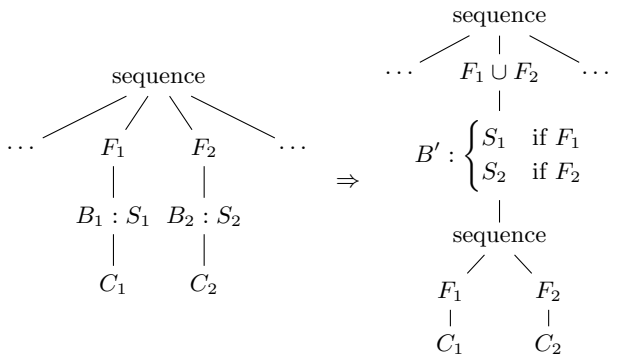

Figure 3: Fuse bands B_1 and B_2

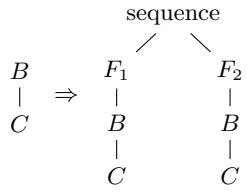

Figure 4: Order the active statement instances at B according to filters F_1 and F_2

Additionally, the schedule spaces of the bands need to be the same since the two schedules will be combined into a single schedule. The two parents are replaced by a single filter node with as filter the union of the filters. The two bands are replaced by a single band with as partial schedule the partial schedules of the original bands on the corresponding filters. The new band has a single sequence child with as children the original two filters with in turn as children the children of the original bands (if any). This transformation is shown schematically in Figure 3.

- Ordering. This operation takes a band node and two filters that partition the active statement instances at the band node as input. The tree is updated such that the elements that satisfy the first filter are executed before the elements that satisfy the second filter. That is, the subtree rooted at the band is duplicated. Each of the filters is inserted in one of the copies of this subtree and subsequently attached as children of a sequence node that replaces the original band. This transformation is shown schematically in Figure 4 and can be used to express a generalized form of loop distribution that allows for the separation of a subset of the instances of a statement.
- Reorder the children of a sequence node.
- Sink a band. This operation moves a band node down to the leaves of the subtree underneath its original position.

This section has shown how to apply each of the transformations of Section 2.2.2 on a schedule tree representation.

Note that we do not allow for the application of affine transformations *across* bands. This restriction can be seen as a form of type safety. Instead, the bands first need to be explicitly combined and/or fused into a single band node, after which the transformation can be applied inside the single band node. In the extreme case all bands are merged into a single band node, in which case the schedule tree essentially degenerates into a union map representation.

5. SCHEDULE TREE USE IN PPCG

In this section, we describe our experience with the use of the schedule trees of Section 4 in `PPCG` [25], which is a polyhedral parallelizing compiler that takes C code as input and produces parallel CUDA code. The main steps involve the extraction of a polyhedral model, dependence analysis, Pluto-like scheduling, the identification of the outermost tilable band, tiling this band, mapping the tile loops to block identifiers and the point loops to thread identifiers, the introduction of transfers to/from shared memory and registers and the generation of an AST.

5.1 Problems in the original implementation

The main complication is the generation of the AST. This AST contains both the kernel code that should be run on the device and the host code that launches these kernels. Generating this AST using a single call to an AST generator is complicated since the host and thread identifiers, which are symbolic constants for the kernel code, should not appear in the host code. Moreover, the copying to/from shared memory and registers and the associated synchronizations need to be inserted in the right position in this AST.

In the original implementation of `PPCG`, these problems were solved through the use of nested AST generation. In particular, an outer call to the AST generator would generate the host code. During the construction of the leaves of this AST, the AST generator would call back to `PPCG`, which would call the AST generator again to generate the outer loops of the kernel code along with the insertion points for the copying code and the synchronizations. A series of third, innermost calls to the AST generator would then generate the inner loops of the core kernel code or the actual copying code or synchronizations.

Although the use of nested AST generation solved the problem at hand, it also made the code very hard to understand and debug. In particular, the piecewise construction of the AST required non-obvious communication between the different stages and problems encountered during inner AST generations were difficult to impossible to reproduce independently since they would depend on the internal state of the outer AST generations.

The main schedule representation used by the original version of `PPCG` is the union map. The band forest produced by the `isl` scheduler was only used to detect the outermost tilable band and then immediately converted to a union map. Because of the requirement of a single schedule space, this map was then padded with zeros with the schedule dimensions mapped to block and threads aligned across the different kernels. Different stages of the schedule with different transformations applied were then maintained to be used in different operations at different levels of the AST generation. Generalizing the implementation to handle split tiling [12] only exacerbated these problems.

5.2 New implementation

The use of schedule trees resolves most of the problems mentioned above. The schedule tree created by the scheduler is incrementally modified to reflect the tiling, the mapping to block and thread identifiers and the introduction of transfers to/from shared memory and registers and is then passed as a single entity to the AST generator. In case of unexpected behavior, this single schedule tree can be investigated in isolation, significantly improving the debugging experience.

In order to be able to express all transformations, we make creative use of the various node types and also extend the domain type. Let us consider the steps in a bit more detail.

- Outermost tilable loops are identified by running a visitor on the tree that checks each band node for suitability of mapping to kernel code. In particular, the band should not only be tilable, it should also have at least one parallel dimension since it will be mapped to block and thread identifiers after tiling. Each time a suitable node has been found, the remaining steps are performed on this node (and its descendants) by making local changes to the schedule tree. A mark node is also introduced before the selected node such that the kernels can be easily identified in the generated AST.

- Tiling the band is a matter of applying one of the operations of Section 4.2. This tiling reduces the memory footprint of the point band, possibly enabling a mapping to shared memory, and splits a single level of parallel loops into two levels of parallel loops.

- Based on some defaults or user choices, taking into account the code inside the current subtree, appropriate grid and block sizes are determined. An extra context node is then inserted before the tile band that introduces symbolic constants corresponding to the block and thread identifiers, along with bounds on their values derived from the grid and block sizes. This context node ensures that conditions on block and thread identifiers that are implied by the grid or block sizes are simplified away and that conditions involving those identifiers are not hoisted beyond this point. The effect of such an internal context node is then very similar to the effect of a nested AST generation phase, except that only a single schedule tree is needed.

- The actual mapping to block and thread identifiers is performed by introducing filter nodes before the tile band and before the point band. These filters select the statement instances that are to be executed by the block (or thread) identified by the symbolic constants introduced by the context node. As always, these symbolic constants have a fixed but unknown value. When the code is actually run on the device, it is executed for each possible value of the block and thread identifiers, ensuring that all statement instances get executed.

- Without going into too much details about how the copying to/from shared memory and registers is performed, it should be clear that we typically want to introduce copies from global memory before a certain loop in the generated AST and copies back to global memory after a certain loop. Theoretically, it would be possible to add copying statement instances to the original domain and to extend all band and filter nodes

between the root and the current node to ensure that the copies are performed in the right place. This is, however, not very convenient or transparent and it also brings in a risk of overfitting. That is, in principle, we only want to perform any copying from global memory if the copied data is actually going to be used, but the additional run-time tests that need to be performed to check if any data is needed may in some cases be more expensive than occasionally copying some data that may not be needed. In particular, the evaluation of the tests may require extra registers, which in turn may result in register spilling and/or a reduction in the number of blocks executed in parallel. (For copying back to global memory, we usually do want to be exact, or at least not copy back any data that we have not copied in first.)

Our solution is to extend the concept of domain nodes to allow for the introduction of statement instances *relative to the current schedule dimensions*. That is, the iteration domain is allowed to refer to the schedule dimensions of the outer band nodes. For example, to generate the copy-in code to bring the data needed by a given subtree, we may apply the concatenation of the outer band schedules to the read access relation to obtain the set of copying instances that need to be executed relative to the current position in the tree. These copying instances can then be introduced by an additional domain node at the current position. In order to reduce the risk of overfitting, we allow the AST generator to generate an AST for the newly introduced instances even if the outer part of the AST generated thus far does not guarantee that at least one of the original statement instances is executed. For example, when copying data from global memory to shared memory, we introduce copying instances not just for the data elements that are needed, but for an entire tile as that usually results in much simpler and more efficient code. Due to the above relaxation, the copying may occasionally also be performed if none of the copied elements are needed. For copying data back to global memory, we only introduce instances corresponding to data elements that have actually been written. The relaxation therefore has no effect in this case.

The only disadvantage compared to nested AST generation is that the extra statements that refer to outer schedule dimensions cannot exploit any potential separation in the outer schedule dimensions. In particular, in case of nested AST generation, if any separation occurs on the host AST, then a separate kernel will be created for each part. Then, each kernel may be simplified in the specific context of that part of the host AST. In the case of schedule trees, however, the kernels are defined on the schedule tree before any AST generation. As a result, AST generation for these kernels cannot take advantage of constraints derived from a potential separation. Instead, in case of separation, the separated loops will launch the exact same kernel for different iterations of the host code. This has not proved to be a problem so far. If it would ever turn out to be a problem, one possible solution would be to apply (part of) the separation phase of the AST generation on the schedule trees themselves. This phase could then be performed before the kernels are defined

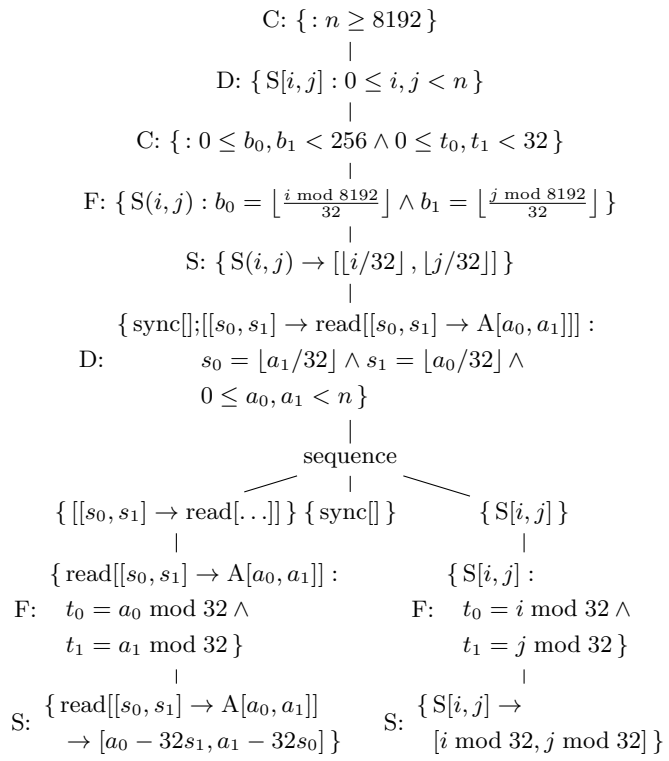

Figure 5: Simple PPCG generated schedule tree for matrix transpose. Node types: Context, Domain, Filter, Schedule. The children of the sequence node are filters

on the schedule trees.

Figure 5 shows a simple PPCG generated schedule tree for a matrix transpose application. The outer context introduces constraints on the parameters ($n \geq 8102$) to simplify the remaining expressions. The inner context introduces block (b_0, b_1) and thread identifiers (t_0, t_1). In this simple example, there are no band nodes in between the two context nodes, but in general such band nodes would produce code to be executed on the host. The filter matches statement iterations to block identifiers and the band node schedules the tile loops. Additional statements are then introduced for reading data into shared memory and for synchronization. The sequence node orders the copying before the synchronization and the core computation. Further filter nodes match statement instances to thread identifiers and the final band nodes complete the schedule.

6. CONCLUSIONS AND FUTURE WORK

By analyzing the situations in which scheduling information is used we showed that most schedules have an inherent tree structure. Yet, we found that existing representations model such tree structure only implicitly, if at all. In this work, we presented the concept of an explicit schedule tree and we propose the use of dedicated tree nodes for important schedule properties. The proposed schedule trees not only make complex schedules easier to modify and read, but with their use in PPCG we also showed that they strongly simplify the implementation of advanced transformation frameworks.

Having currently only converted the core of `PPCG`, we plan to replace the union map schedules in `pet` [24], `iscc` [23] and in the `isl` dataflow analysis by schedule trees. As explained in Section 5, it may also be interesting to perform (part of) the separation phase of AST generation on the schedule trees. This may also avoid the need for "reentrancy". We may also introduce additional types of nodes or extensions to represent for example parametric tiling.

We are also considering embedding some form of clustering in schedule trees. During part of the (manual or automatic) scheduling process, we may want to handle groups of statements in the same way. It may be useful to express this grouping directly in the schedule trees, possibly through compression and expansion nodes that map several statement instances to a single instance or vice versa. These nodes should have the same benefit as the sequence nodes, i.e., explicitly making information available so that it does not have to be rediscovered and that it can easily be read off by the user.

Acknowledgments. This work is partly funded by the European FP7 project CARP id. 287767.

7. REFERENCES

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[2] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2004. IEEE Computer Society.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, 2008.

[4] C. Chen. Polyhedra scanning revisited. In *PLDI*, 2012.

[5] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, June 2008.

[6] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming*, 24, December 1996.

[7] A. Darte, Y. Robert, and F. Vivien. Loop parallelization algorithms. In *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques and Run Time Systems*, volume 1808 of *Lecture Notes in Computer Science*, pages 141–171. Springer Verlag, 2001.

[8] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. of Parallel Programming*, 20(1), 1991.

[9] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec. 1992.

[10] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model. Springer-Verlag, 1996.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3), June 2006.

[12] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for gpus: automatic parallelization using trapezoidal tiles. In *GPGPU-6*. ACM, 2013.

[13] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[14] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Dept. of CS, Univ. of Maryland, 1996.

[15] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *IEEE First International Conference on Algorithms and Architectures for Parallel Processing (ICAPP 95)*, volume 1, Apr. 1995.

[16] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997.

[17] V. Maslov. Lazy array data-flow dependence analysis. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*. ACM Press, 1994.

[18] L.-N. Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, Jan. 2010.

[19] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16, July 1994.

[20] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, R. Upadrasta, et al. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, 2010.

[21] N. T. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université Paris Sud XI, Orsay, September 2007.

[22] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*. Springer, 2010.

[23] S. Verdoolaege. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[24] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[25] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013.

[26] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. 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*, 2012.