Live-Range Reordering

Sven Verdoolaege
Polly Labs and KU Leuven
sven.verdoolaege@gmail.com

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

ABSTRACT
False dependences are caused by the reuse of memory to store different data. These false dependences severely constrain the schedule of statement instances, effectively serializing successive accesses to the same memory location. Several array expansion techniques have been proposed to eliminate some or all of these false dependences, enabling more reorderings of statement instances during loop nest transformations. However, array expansion is only relevant when complemented with a storage mapping optimization step, typically taking advantage of the fixed schedule set in earlier phases of the compilation, folding successive values into a compact set of contracted arrays. Furthermore, array expansion can result in memory footprint and locality damages that may not be recoverable through storage mapping optimization when intermediate transformation steps have abused the freedom offered by the removal of false dependences. Array expansion and storage mapping optimization are also complex procedures not found in most compilers, and the latter is moreover performed using suboptimal heuristics (particularly in the multi-array case). Finally, array expansion may not remove all false dependences when considering data-dependent control and access patterns. For all these reasons, it is desirable to explore alternatives to array expansion as a means to avoid the spurious serialization effect of false dependences. This serialization is unnecessary in general, as semantics preservation in presence of memory reuse only requires the absence of interference among live-ranges, an unordered constraint compatible with the their commutation. We present a technique to deal with memory reuse without serializing successive uses of memory, but also without increasing memory requirements or preventing important loop transformations such as loop distribution. The technique is generic, fine-grained (instancewise) and extends two recently proposed, more restrictive approaches. It has been systematically tested in PPG and shown to be essential to the parallelizing compilation of a variety of loop nests, including large PENCIL programs with many scalar variables.

1. INTRODUCTION AND MOTIVATION
Polyhedral compilation is a framework for analyzing and transforming program fragments that are "sufficiently regular" through a mathematical abstraction that models the individual statement instances and array elements using a compact representation. When using polyhedral compilation to transform a sequential program, the accesses in the program are first analyzed to compute dependences between statement instances that restrict the possible reorderings of these instances. In particular, a statement instance that depends on another statement instance needs to be executed after that other statement instance.

Roughly speaking, dependences come in two types: the true or flow dependences, i.e., those that are strictly needed because some data produced and stored in memory by one statement instance is (or may be) used by another, and the false dependences, i.e., those that are caused by the reuse of memory to store different data and that only serve to ensure that data is not overwritten between production and use. The true dependences will also be called live-ranges because they correspond to a pair of statement instances between which some memory location may be live. While it is more customary to only consider live-ranges that end in the last use, the last use is only known after scheduling and therefore all uses need to be considered during scheduling.

Consider for example the slightly contrived code in Listing 1. An element of the $t$ array is used to communicate data from instances of statement $S_1$ to the immediately following instances of statement $S_2$, constituting live-ranges between these pairs of instances. However, the same memory location is overwritten by some subsequent instances of $S_1$ as well as by some instances of statement $S_3$. One way of handling such memory reuse is to introduce false dependences between instances of $S_2$ and all later instances of both $S_1$ and $S_3$ that access the same element. This ensures that the live-ranges will not overlap in the transformed code, but it does so by fixing a particular execution order of those live-ranges, preventing the loop nests in Listing 1 from being completely fused and tiled. The two nests can still be tiled separately in this case by first applying a skewing transformation. If the $t$ array is replaced by a single scalar, then these false dependences even prevent any fusion or tiling. A source-to-source polyhedral compiler such as Pluto+ (Bondhugula et al. 2008; Acharya and Bondhugula 2015) will then also refuse to perform such fusion or tiling.

An alternative to the introduction of false dependences is to convert the program to single assignment form by applying array expansion (Feautrier 1988). Transformations
void f(int n, int A[restrict static n][n],
   int B[restrict static n][n],
   int C[restrict static n][n])
{
    int t[2 * n - 1];

    #pragma scop
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j) {
        S1: t[i + j] = A[i][j];
        S2: C[i][j] = t[i + j];
      }

    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j) {
        S3: t[i + j] = B[i][j];
        S4: C[i][j] += t[i + j];
      }
    #pragma endscop
}

Listing 1: Illustrative example

can then be performed only taking into account the true
dependences. The array expansion may incur a prohibitive
increase in the memory requirements, but some of this in-
crease can be removed again by applying array contraction
techniques (see, e.g., Wilde and Rajopadhye 1996; Darte et
al. 2005; and their references). The result of array contrac-
tion may even require less memory than the original. Still,
the increased freedom introduced by the array expansion
can cause some of the live ranges that were originally us-
ing the same memory to overlap. This overlap of live-ranges
prevents the expanded memory location from being mapped
to the same memory after transformation and the memory
requirements are still increased. As a simple example, ex-
ansion allows the compiler to fuse the loop nests in Listing 1
(after performing a loop interchange on one of the nests) and
to execute an instance of S2 between the corresponding in-
stances of S3 and S4 in order to bring the two accesses to
C as close as possible to each other. However, this execu-
tion order prevents the same memory location from being
used to store the values produced by S1 and S3, resulting
in an overall increase in memory requirements (compared to
the case where array contraction is applied to the original
(1)) illustrate that this failure to contract does happen in
practice and that it can have a significant impact on per-
formance. Note that basic array expansion requires exact
dataflow analysis. If the input program does not allow such
an analysis to be performed exactly, then more advanced
techniques need to be used that incur further increases in
memory requirements (Vanbroeckhoven et al. 2005) or the
expansion needs to be limited to maximal static expansion
(Barthou et al. 2000), in which case some false dependences
remain. It is also possible to postpone the expansion
until after a schedule has been computed that ignores the
false dependences (Trifunovic and Cohen 2010). The latter
approach, however, is only limited in scope, but a limited form of ex-
ansion is still required in general.

Polyhedral compilers such as GCC/Graphite (Trifunovic,
Cohen, et al 2010) and LLVM/Polly (Grosser, Gräßlinger,
et al. 2012) that operate on the compiler internal representa-
tion typically do so after a conversion to static single as-
signment (of the scalars) and consider basic blocks as indi-
visible units. Scalars that are only used within a given basic
block are usually not modeled at the polyhedral level by
such compilers. These scalars then do not give rise to false
dependences. For example, S1 and S2 in Listing 1 belong to
the same basic block and so do S3 and S4. If the t array were
to be replaced by a scalar, then no false dependences would
therefore be generated based on the two single assignment
scalars originating from t. However, this mechanism only
applies to accesses to scalars, it easily breaks down when
the basic blocks get split for whatever reason and the indi-
visibility of basic blocks prevents optimizations such as loop
distribution that require different parts of a basic block to
be transformed differently. In theory, it would be possible to
extend this scheme and treat the entire compound statement
that accesses any given reused variable as an indivisible unit,
but then the loops inside the compound statement could no

Listing 2: Code from Listing 1 after fusion and tiling

```c
for (c0=0; c0<n; c0+=32)
  for (c1=0; c1<n; c1+=32)
    for (c2=0; c2<=min(31,n-c0-1); c2+=1)
      for (c3=0; c3<=min(31,n-c1-1); c3+=1) {
        t[c0+c1+c2+c3]=A[c1+c3][c0+c2];
        C[c1+c3][c0+c2]=t[c0+c1+c2+c3];
        t[c0+c1+c2+c3]=B[c0+c2][c1+c3];
        C[c1+c3][c0+c2]+=t[c0+c1+c2+c3];
      }
```

```
Listing 1: Illustrative example
```
nor basic block clustering. It therefore does not risk increasing memory requirements and does allow loop distribution. No claim is made that this category of approaches is always the right choice. However, within this category, the presented technique is more fine-grained than that of Baghdadi (2011) and more general than those of Baghdadi, Cohen, et al. (2013) and Mehta (2014) Chapter 5. In particular, it is not restricted to loop tiling or limited forms of loop fusion. Although in practice the technique is mostly useful for dealing with reuse of scalars, it can handle reuse of array elements equally well. For example, it allows the code in Listing 1 to be fused and tiled to produce the code in Listing 2. The main idea is to use the criterion of Baghdadi, Cohen, et al. (2013) during the search for enabling affine transformations by introducing and handling conditional validity constraints that prevent live-ranges from overlapping. An initial implementation has been publicly available in PPCG since version 0.02 (April 2014), but it has only been extensively tested through the experiments of Baghdadi, Beaugnon, et al. (2015). This testing uncovered some issues that have been fixed in version 0.04 (June 2015).

2. BACKGROUND

This section presents some terminology and known results that are useful for describing the live-range reordering technique.

2.1 PPCG

PPCG is a source-to-source polyhedral compiler that takes sequential C or PENCIL (Baghdadi, Beaugnon, et al. 2015) code as input and that produces CUDA or OpenCL code as output. It was originally described by Verdoolaege, Juega, et al. (2013) and further extended by Verdoolaege (2015), mainly to support extra features required by the PENCIL language. PPCG relies on pet (Verdoolaege and Grosser 2012) to extract a polyhedral model and on isl (Verdoolaege 2010) to perform dependence analysis, scheduling and AST generation (Grosser, Verdoolaege, et al. 2015). The terminology described below is as used by isl and PPCG. The live-range reordering technique has also been implemented partly in isl and partly in PPCG.

2.2 Polyhedral Model

A polyhedral model (Feautrier and Lengauer 2011) is a compact mathematical abstraction of a program fragment. For the purpose of this paper, a polyhedral model consists of an instance set, access relations, dependence relations and a schedule. The instance set describes the dynamic execution instances in the program fragment. Taking statements as units of execution, the instance set of the program fragment in Listing 1 can be described as

\[
\{ S1[i, j] : 0 \leq i, j < n; S2[i, j] : 0 \leq i, j < n; S3[i, j] : 0 \leq i, j < n; S4[i, j] : 0 \leq i, j < n \},
\]

with n a constant symbol representing the value of n, which remains fixed throughout the execution. The unit of execution in pet may also be larger than an individual statement, in particular if some of these statements contain dynamic control (Verdoolaege 2015).

The access relations describe which data elements are accessed by each statement instance. A distinction is made between reads and writes. Since it may not be possible to determine at compile-time exactly which data elements will be accessed at run-time, or it may not be possible to represent the accesses exactly, these reads and writes may be overapproximations and are called may-reads and may-writes. The must-writes form a subset of the may-writes that are known to be performed with certainty. This leads to three access relations, the may-read access relation, the may-write access relation and its subset, the must-write access relation. An access relation is an instance of a binary relation, containing pairs of elements. The set of all elements that appear in the first position is called the domain of the relation. The set of all elements that appear in the second position is called the range of the relation. In the case of an access relation, the domain is the set of statement instances that access one or more data elements and the range is the set of data elements that are being accessed. For example, the may-read access relation (modulo domain constraints) for the program in Listing 1 is

\[
\{ S1[i, j] \rightarrow A[i, j]; S2[i, j] \rightarrow t[i + j]; S3[i, j] \rightarrow B[i, j]; S4[i, j] \rightarrow C[j, i]; S4[i, j] \rightarrow t[i + j] \},
\]

while the may-write access relation, here equal to the must-write access relation, is

\[
\{ S1[i, j] \rightarrow t[i + j]; S2[i, j] \rightarrow C[i, j]; S3[i, j] \rightarrow t[i + j]; S4[i, j] \rightarrow C[j, i] \}.
\]

A special kind of access relation is formed by what are known as the kills. The meaning of a kill is that no values can possibly flow through the “accessed” data elements. The domain of the kills is formed by instances of additional statements called kill statements. Kill statements are introduced automatically for locally defined scalars or arrays, one at the point of the declaration and one at the point where the declaration goes out of scope. The accessed data elements are formed by the scalar or the entire array. Kills can also be introduced explicitly through a call to \_pencil\_kill (Verdoolaege 2015). For example, if the entire body of the function in Listing 1 is considered (rather than just the marked part), then two additional statements are introduced, one at the start and one at the end, both killing the entire t array.

A dependence relation contains pairs of elements where the second depends on the first for its execution. There are several different dependence relations and they are all derived from the access relations as explained in Section 2.3.

If a statement accesses the same array multiple times, i.e., through multiple references, then for some applications, it may not be sufficient to know which statement instance performs the access and instead the access needs to be related to a particular reference inside the statement instance. For example, when PPCG is determining which data to copy to/from a device, it checks which of the write references produce data that is only used inside a given kernel. This requires the reference to be identifiable from the dependence relations, which in turn requires them to be encoded in the access relations. In pet, reference identifiers are added to the domain of the access relations to form tagged access relations. For example, assuming the individual array references are called R0, R1, . . . R7, the tagged may-read access relation of the program in Listing 1 would be

\[
\{ S1[i, j] \rightarrow R1[i, j]; S2[i, j] \rightarrow R3[i, j]; S3[i, j] \rightarrow R5[i, j]; S4[i, j] \rightarrow R6[j, i]; S4[i, j] \rightarrow R7[i, j] \rightarrow t[i + j] \}.
\]
2.3 Dependence Analysis

Dependence analysis determines which statement instances depend on which other statement instances. A distinction is typically made between memory-based dependence analysis and value-based dependence analysis (Pugh and Wonnacott 1994). The latter is also called dataflow analysis (Feautrier 1991). In (memory-based) dependence analysis, a statement instance is considered to depend on any previous statement instance that accesses the same data element at least one of those two accesses is a write. In dataflow analysis, a statement instance performing a read is considered to depend on the last preceding statement instance that performs a write to the same data element. An additional result of dataflow analysis is the set of reads for which there are no preceding writes.

The dependence analysis procedure in isa1 generalizes over these two extremes. In particular, it takes a sink access relation, a may-source access relation, and a schedule as input and determines for each domain element of the must-source access relation that is executed before i (according to the schedule) and that access the same data element, as well as all intermediate domain elements of the may-source access relation that access this same data element. If there is no such domain element in the must-source access relation, then the procedure collects all previous domain elements of the may-source access relation that access this same data element. In other words, starting from the point in the schedule where the source domain element is executed, the procedure collects all previously executed domain elements of the may-source and must-source relations that access the same data element until an element of the must-source is reached. The sinks for which no such must-source is encountered are collected as well.

The traditional memory-based dependences are obtained by running the procedure twice, once with as may-source the may-write access relation and as sink the may-read access relation and once with as may-source the union of the may-reads as sinks and may-sources and as source those induced by t are dashed). The statement instances are represented as $S1: \bullet, S2: \circ, S3: \oplus, S4: \square$. The circled instances on the left perform a live-in access. The circled instances on the right perform a live-out access.

The tagged may-reads as sinks, the tagged may-writes as may-sources and the union of the tagged must-writes and the tagged kills as must-sources. The domain of the tagged kills is subsequently removed from the result. The difference forms what are called the tagged flow dependences. That is, there is a flow dependence from a may-write to a later may-read as long as there is no intermediate must-write or kill. The sinks for which no corresponding must-source is found during the dependence analysis are considered to be the live-in accesses. That is, the live-in accesses are the may-reads that may read a value that was written before the start of the program fragment under analysis. For example, the code in [Listing 1] has the following flow dependences (modulo domain constraints):

$$\{ S1[i, j] \rightarrow S2[i, j]; S3[i, j] \rightarrow S4[i, j]; S2[i, j] \rightarrow S4[j, i] \}$$

(5)

These are shown on the left of Figure 1. The live-in accesses (modulo domain constraints) are as follows:

$$\{ S1[i, j] \rightarrow A[i, j]; S3[i, j] \rightarrow B[i, j] \}$$

(6)

The second application takes the may-writes as sinks, the must-writes as must-sources and the union of the may-reads and the may-writes as may-sources. The resulting dependences are called the false dependences. Those with a may-read as source are also known as anti-dependences, while those with a may-write as source are also known as output dependences. Setting must-sources is not strictly necessary, but it allows for the removal of some transitively covered
false dependences that would otherwise be redundant. The code in Listing 1 has the following anti-dependences (modulo domain constraints):

\[
\begin{align*}
S2[i, 0] & \rightarrow S3[0, i]; S2[n - 1, j] \rightarrow S3[j, n - 1]; \\
S2[i, j] & \rightarrow S1[i + 1, j - 1]; S4[i, j] \rightarrow S3[i + 1, j - 1].
\end{align*}
\]

and the following output dependences (modulo domain constraints):

\[
\begin{align*}
S2[i, j] & \rightarrow S4[j, i]; \\
S1[i, 0] & \rightarrow S3[0, i]; S1[n - 1, j] \rightarrow S3[j, n - 1]; \\
S1[i, j] & \rightarrow S1[i + 1, j - 1]; S3[i, j] \rightarrow S3[i + 1, j - 1].
\end{align*}
\]

Their union is shown on the right of Figure 1.

The third application takes as may-sources the tagged may-writes and as sinks the union of the tagged must-writes and the kills. The domain of the resulting dependences consists of the pairs of statement instances and reference identifiers that access elements that are definitively overwritten or killed. Specializing to the shared array elements results in dependences, while others may not. The inputs are called schedule constraints and they come in three groups: the validity constraints, the proximity constraints and the coincidence constraints. The validity constraints determine which statement instances need to be scheduled after which other statement instances. The proximity constraints determine which statement instances should be scheduled as close as possible to which other statement instances. The coincidence constraints determine which pairs of statement instances should be scheduled together by as many of the outer members in a band as possible. The coincidence constraints also determine whether a band member is marked coincident, i.e., parallelizable. The main difference between proximity constraints and coincidence constraints is that the former care about the distances between pairs of statement instances, while the later only care about whether these distances are zero or not. Since a user may only care about some pairs of instances being scheduled together and not about them being scheduled close to each other when they cannot be scheduled together, it can be useful in some cases to add some dependences to the coincidence constraints, but not to the proximity constraints. Similarly, because of the effect on the coincident property, it may be equally useful to add some dependences to the coincidence constraints, but not to the coincidence constraints. These two cases will be illustrated in Section 3.3.

A band is constructed one member at a time. In particular, in each iteration, an affine function is constructed for each statement that is linearly independent of all previously computed members in the band as well as all members of outer bands. Furthermore, the affine function is constructed such that it respects all the validity constraints, meaning that the second element in a pair of elements is assigned a value that is greater than or equal to the value assigned to the first element, and such that the largest differences between these values over all the proximity dependences is minimized. If there are any coincidence constraints, then this difference is initially set to zero over all the coincidence constraints. If this fails to produce a solution, then the coincidence constraints are dropped for the purpose of constructing the current band and another attempt is made at constructing the next affine function. Once a band is completed, i.e., when no more such affine functions can be found, all dependences that are scheduled apart by the band are removed. That is, only those pairs of elements are kept that are assigned the same value by the entire band. If the construction of a band fails for whatever reason, then a single-member band is constructed that covers as many dependences as possible by applying one step of a Feautrier-like scheduler (Feautrier 1992). When a sufficient total number of band members have been computed (greater than or equal to the dimension of the instance set restricted to each statement), any remaining validity dependences are handled by topologically sorting the statements according to those dependences.

When live-range reordering is disabled, PPCG uses the union of the flow and the false dependences as validity, proximity and coincidence constraints.

2.5 Permutability

Since the members in a band are computed with respect to the same set of (validity) dependences, they are fully per-
mutable. The band can therefore also be tiled. As explained above, the false dependences (which are included in the validity dependences) ensure that live-ranges will not overlap. However, in case of memory reuse in the input program, these false dependences essentially enforce a serialization of the corresponding live-ranges such that no (non-trivial) permutable bands can be constructed. While sufficient for constructing a valid schedule, this serialization is by no means necessary.

Baghdadi, Cohen, et al. (2013) propose a permutability criterion that essentially allows live-ranges to be arbitrarily reordered within a band as long as those live-ranges are local to the band, where a live-range is considered to be local to a band if both elements in the live-range are assigned the same values by the members of the band. In particular, an anti-dependence can be ignored if it only serves to keep pairs of live-ranges apart that are both local to the band. The live-ranges that an anti-dependence is meant to keep apart are called adjacent to the anti-dependence. That is, a live-range and an anti-dependence are considered to be adjacent to each other if the source of one is the sink of the other. Consider, for example, a band with an identity schedule for the statements in the first loop of the code in Listing 1, i.e., \( \{ S1[i,j] \rightarrow [i,j]; S2[i,j] \rightarrow [i,j] \} \). All the live-ranges in \( \{ \) of the form \( S1[i,j] \rightarrow S2[i,j] \) are local to this band because both sides are assigned the same values. The anti-dependence \( S2[0,1] \rightarrow S1[1,0] \) is adjacent to the live-ranges \( S1[0,1] \rightarrow S2[0,1] \) and \( S1[1,0] \rightarrow S2[1,0] \). Since both these live-ranges are local to the band, the anti-dependence can be ignored. Note that in order to account for both changes in the schedule and the permutations themselves, all anti-dependences need to be considered and not only those that are not transitively covered. That is the anti-dependences of \( [i,j] \) need to be replaced by

\[
\begin{align*}
S2[i,j] &\rightarrow S1[i',j'] : i + j = i' + j' \land i' > i; \\
S2[i,j] &\rightarrow S3[i',j'] : i + j = i' + j'; \\
S4[i,j] &\rightarrow S3[i',j'] : i + j = i' + j' \land i' > i.
\end{align*}
\]

Output dependences are usually covered by a pair of a live-range and an anti-dependence and these can therefore also be ignored. The only exceptions are those for which the first write in the pair of writes has no corresponding reads and those for which the value of the second write is still live after the program fragment under consideration. In the example, there are no output dependences of the first kind, but all those output dependences where the second write is live-out \( [i,j] \) are of the second kind, i.e.,

\[
\begin{align*}
S2[i,j] &\rightarrow S4[i,j]; \\
S1[i,j] &\rightarrow S3[i+j,0]; S1[i,j] \rightarrow S3[n-1,i+j-n+1]; \\
S3[i,j] &\rightarrow S3[i+j,0]; S3[i,j] \rightarrow S3[n-1,i+j-n+1].
\end{align*}
\]

Note that here as well transitively covered output dependences should not be removed.

Baghdadi, Cohen, et al. (2013) apply their relaxed permutability criterion after the construction of a schedule has been computed that does take into account the false dependences, by checking if the criterion allows nested bands to be considered as a single combined band. The following section explains how the criterion can be used during the construction of the schedule.

### 3.2 Conditional Validity Constraints

In order to support live-range reordering, the scheduler is extended to support an additional type of schedule constraints called conditional validity constraints. Unlike the other schedule constraints, a conditional validity constraint does not consist of a single binary relation, but of two binary relations, the condition and the conditioned validity constraint. These two relations may be either both tagged or both untagged. The tags, if present, are completely arbitrary. That is, they may be reference identifiers as in Section 2.2 but they may also represent accessed array elements as suggested in Section 3.1 or even something else entirely.

The meaning of a conditional validity constraint is as follows. If a given band does not assign the same values to domain \( i \) and range \( j \) of an element of the condition relation, then any element of the conditioned validity constraint relation that has \( j \) as domain or \( i \) as range (i.e., that is adjacent to the element \( i \rightarrow j \)) needs to be treated as a regular validity constraint. That is, the band needs to either make the conditions local or it needs to satisfy the corresponding conditioned validity constraints. The tags, if present, are only used to determine which elements of the condition relation are adjacent to which elements of the conditioned validity constraint relation. That is, they are ignored for the purpose of evaluating the band members.

Clearly, the intended use of conditional validity constraints in case of live-range reordering is for the conditions to be set to the live-ranges and the conditioned validity constraints to be set to the anti-dependences. In the running example, the
1  Coincidence ← false
2  while band not full-dimensional do
3      ConstructAffineScheduleFunction(Coincidence)
4      if no solution then
5          if Coincidence then
6              Coincidence ← false
7              continue
8          else
9              break
10     add schedule function to band
11     V ← violated conditioned validity constraints
12     C ← condition constraints adjacent to V
13     L ← domain and range of C elements co-scheduled
14     mark C as local
15     if not L then
16         clear current band
17         Coincidence ← true
18     if band is empty then
19         return Feautrier()
20     return current band

Algorithm 1: Schedule band construction

live-ranges of \( \{5\} \) would be used as the conditions while the anti-dependences of \( \{10\} \) would be used as the conditioned validity constraints. This ensures that a live-range is either local to the band being constructed or that all adjacent anti-dependences are satisfied. In PPCG, the tagged versions of these dependences are used. The details are explained in Section 3.3 below.

The changes required to the scheduling procedure of Section 2.4 are relatively minor. During the construction of a band, the conditional validity constraints are initially ignored. Each time a band member has been computed, the conditioned validity constraints are checked for violations. If there are any, then the adjacent elements of the condition relation are forced to be local. That is, domain and range of these elements are required to be assigned the same value by any subsequent members in the band. If, furthermore, these pairs of elements are not assigned the same values by the current member or any previously computed member of the band, then the computation of the band is restarted from scratch. The only difference with the previous attempt to compute a band is that all the elements of the condition relation that were forced to be local, remain in this state. In particular, if the coincidence constraints had been dropped during the current attempt to compute a band, then they are reconsidered in the next attempt. In practice, it is not individual elements of the condition constraint that are marked local, but entire groups satisfying the same conjunction of constraints. Since there are only a finite number of such groups and since each reset marks at least one additional group as local, the process is guaranteed to terminate. Algorithm 1 shows a schematic overview of the schedule band construction algorithm. The code from Line 11 until Line 17 deals with conditional validity constraints.

If the scheduler needs to resort to a step of the Feautrier-like scheduler, then the conditioned validity constraints are treated in the same way as regular validity constraints during this step.

### 3.3 Live-range Reordering

Essentially, in order to enable live-range reordering, the false dependences need to be removed from the validity constraints and replaced by conditioned validity constraints with the live-ranges as conditions. However, by its very nature, live-range reordering may change the order in which live-ranges are executed. As explained in Section 2.3, this means that the user can no longer assume that transitively covered dependences remain covered if any link in the chain of covering dependences is a live-range. In other words, the conditioned validity constraints need to include the full set of anti-dependences. Care also needs to be taken with respect to the output dependences and the coincidence constraints. This section describes in some detail how these issues are handled in PPCG.

When live-range reordering is enabled, PPCG computes tagged flow dependences (i.e., tagged live-ranges), false dependences and live-out accesses as before. In addition, it also computes order dependences and forced dependences, as explained below. (There is a minor difference in how “independences” (Verdoolaege 2015) are taken into account during the computation of the flow dependences, but this is beyond the scope of the present paper.)

The (tagged) order dependences will be used to prevent live-ranges from overlapping. They are computed by taking as sinks the tagged may-writes and as sources the union of the tagged may-read and the (tagged) unmatched writes. The latter are the writes that do not appear in the domain of the tagged flow dependences. That is, the order dependences contain all anti-dependences as well as all output dependences that are not covered by a combination of a live-range and an anti-dependence.

The forced dependences consist of all the validity dependences (other than the live-ranges themselves) that should be enforced even with live-range reordering enabled. These consist of two parts, “external” false dependences and order dependences between flow dependence sources with the same sink. The external false dependences are those that ensure that the live-in accesses remain live-in and similarly for the live-out accesses. They are computed using two applications of the dependence analysis procedure, one with as may-sources the live-in accesses and as sinks the may-writes and one with as sinks the live-out accesses and as may-sources the may-writes. The resulting dependences ensure that no may-write gets moved before a live-in access or after a live-out access to the same memory element.

The order dependences between flow dependence sources with the same sink are needed to ensure that all potential sources of a given sink are executed in the same order as in the input program. This is needed because it is not clear at compile-time which of these potential sources will write the value that is read by the sink and this value should not be overwritten by a value that was written beforehand in the input program. These order dependences are computed using a final application of the dependence analysis procedure. In this application, the may-sources and the sinks are set to the same relation and it is one that has the flow dependence sources as domain and that “accesses” a pair that consists of one of the corresponding flow dependence sinks and an array element accessed by the flow dependence source. The resulting dependences are then pairs of flow dependence sources that share a combination of sink and accessed array element. All the dependences computed above are used in the sched-
ule constraints as follows. The validity constraints are set to the union of the flow and the forced dependences. The proximity constraints are set to the union of the flow dependences, the forced dependences and the order dependences that are derived from array accesses. The conditions of the conditional validity constraints are set to the tagged flow dependences, while its conditioned validity constraints are set to the entire set of tagged order dependences. In principle, the entire set of order dependences would have to be added to the coincidence constraints as well, but PPCG has special support for privatizing scalars. There is no such support at this point for privatizing arrays, which is why their order dependences are still added to the coincidence constraints.

Summarizing the validity of the approach, the live-ranges are preserved by ensuring that no other write gets scheduled between the write w and the read r of a live-range. If only the write or the read is part of the program fragment, then the external false dependences take care of this. Otherwise, a second write w′ that is executed before the write in the original program is prevented from moving between the write and the read as follows. If there are other reads in between the two writes, then w′ forms a live-range with one of these reads, which in turn has an anti-dependence with w. The conditional validity constraints ensure that either the two live-ranges are local or that the anti-dependence is satisfied. If there are no reads in between, then w′ is either unmatched or it also forms a live-range with r. The first case is taken care of by the order dependences, the second by the order dependences between flow dependence sources. A second write w′ that is executed after the read in the original program is prevented from moving between the write and the read because if forms an anti-dependence with the read.

4. EXAMPLES

4.1 Running Example

Let us first consider the example code in Listing 1. Running PPCG (version ppcg-0.04-26-gf956ffe) with the options --target=mc --autodetect --tile produces the code in Listing 2 (with some white-space editing). The corresponding schedule (prior to tiling) is

\[
\begin{align*}
S1[i, j] &\rightarrow [i, j]; S2[i, j] \rightarrow [i, j]; \\
S3[i, j] &\rightarrow [i, j]; S4[i, j] \rightarrow [i, j]; \\
\end{align*}
\]

with a topological sort of the statements inside this band as follows: S1, S2, S3, S4. The corresponding dependence distances of the flow dependences in \(\square\) are \((0, 0)\). The --autodetect option is needed to make pet consider the entire function body, allowing it to add kills to t. This in turn allows PPCG to remove the accesses to t from the live-out accesses as explained near the end of Section 2.3. Without this option, the scheduler needs to ensure that the circled S3 instances on the right of Figure 1 remain the last to access the corresponding element of t, requiring an additional skewing and resulting in a schedule of the form

\[
\begin{align*}
S1[i, j] &\rightarrow [j, i + j]; S2[i, j] \rightarrow [j, i + j]; \\
S3[i, j] &\rightarrow [i, i + j]; S4[i, j] \rightarrow [i, i + j]. \\
\end{align*}
\]

Note that this schedule still allows the two loop nests to be fused completely. That is, the dependence distances over the flow dependences are still \\{(0, 0)\}. However, some of the initial tiles are no longer full tiles.

Using polycc from Pluto+ (version 0.11.3-238-gf4d02e5) with options --pet --maxfuse --tile results in the schedule

\[
\begin{align*}
S1[i, j] &\rightarrow [i + j, i]; S2[i, j] \rightarrow [i + j, i]; \\
S3[i, j] &\rightarrow [i + j, i + n]; S4[i, j] \rightarrow [i + j, i + n]. \\
\end{align*}
\]

prior to tiling. Note that this schedule shifts the second loop nest beyond the first loop nest in the inner dimension, such that the executions of the two loop nests merely alternate, but do not overlap. In particular, the dependence distances over the flow dependences are now \\{(0, x) : 0 \leq x < 2n\}. Reevaluating this schedule using the criterion of Baghdadi, Cohen, et al. [2013] does not have any effect because the schedule is already tiable. As already mentioned in Section 4, the technique of Mehta (2014, Chapter 5) would also not have any effect since the dependence pattern does not fit the special case it handles. Running PPCG with the --no-live-range-reordering option results in the schedule

\[
\begin{align*}
S1[i, j] &\rightarrow [i + j, -j]; S2[i, j] \rightarrow [i + j, -j]; \\
S3[i, j] &\rightarrow [i + j, i]; S4[i, j] \rightarrow [i + j, i]. \\
\end{align*}
\]

prior to tiling, with dependence distances \\{(0, x) : 0 \leq x < 2n - 2 \wedge x \mod 2 = 0\}. Consider now an input program that is identical to the code in Listing 1 except that the t array has been replaced by a scalar. In this case, PPCG produces the schedule in (12) (with or without --autodetect). polycc produces

\[
\begin{align*}
S1[i, j] &\rightarrow [i, j]; S2[i, j] \rightarrow [i, j]; \\
S3[i, j] &\rightarrow [i + n, j]; S4[i, j] \rightarrow [i + n, j], \\
\end{align*}
\]

with dependence distances \\{(0, 0); (x, n - x) : 0 < x < 2n\}. That is, it essentially produces the original schedule and refuses to tile the two loop nests (due to the false dependences). Reevaluating this schedule using the criterion of Baghdadi, Cohen, et al. [2013] would turn the schedule above into a permutable band, which would allow tiling the band, but the two loop nests would still not be effectively fused. The technique of Mehta (2014, Chapter 5) would allow effective fusion, but only at the outer level and not at the inner level.

4.2 Example from Mehta (2014)

Consider now the example program of Mehta (2014, Figure 5.4 (a)), reproduced in Listing 3 except that the lower bound on the k1-loop has been changed from 0 to 1 to avoid an out-of-bounds access in statement S2. Mehta (2014, Figure 5.4 (c)) shows that his technique is only able to fuse the outer two loops of these two loop nests. PPCG (with the same options as before), on the other hand, is capable of fusing all three loops with schedule

\[
\begin{align*}
S1[i, j, k] &\rightarrow [i, j, k]; S2[i, j, k] \rightarrow [i, j, k]; \\
S3[i, j, k] &\rightarrow [i, j, k]; \\
S4[i, j, k] &\rightarrow [i, j, k + 1]; S5[i, j, k] \rightarrow [i, j, k + 1], \\
\end{align*}
\]

and internal topological sort of the statements: S1, S2, S3, S4, S5. Note that if the resulting fused loop is taken as input, then the same technique is also capable of distributing this loop (depending on the optimization criteria) to produce the code in Listing 3.
void foo(int Nx, int Ny, int Nz, 
    int a[restrict static Nx][Ny][Nz], 
    int x[restrict static Nx][Ny][Nz], 
    int rho[restrict static Nx][Ny][Nz]) 
{
    int a0, am1;

    for (int i1 = 0; i1 < Nx; i1++) {
        for (int j1 = 0; j1 < Ny; j1++) {
            for (int k1 = 1; k1 < Nz; k1++) {
                S1: a0 = a[i1][j1][k1];
                S2: am1 = a[i1][j1][k1-1];
                S3: x[i1][j1][k1] = a0 + am1;
            }
        }
    }

    for (int i2 = 0; i2 < Nx; i2++) {
        for (int j2 = 0; j2 < Ny; j2++) {
            for (int k2 = 0; k2 < Nz - 1; k2++) {
                S4: a0 = a[i2][j2][k2];
                S5: rho[i2][j2][k2] = a0 + (x[i2][j2][k2+1] - x[i2][j2][k2]);
            }
        }
    }
}

Listing 3: Example program from Mehta (2014, Figure 5.4 (a))

distribution would not be possible. On the original input, polycr produces
\[ S1[i,j,k] \rightarrow [i,j,k]; S2[i,j,k] \rightarrow [i,j,k]; \]
\[ S3[i,j,k] \rightarrow [i,j,k]; S4[i,j,k] \rightarrow [N_x + i,j,k + 1]; \]
\[ S5[i,j,k] \rightarrow [N_x + i,j,k + 1]. \] (18)
i.e., no fusion, and also refuses to tile the loop nests. (Note that Pluto+ does not appear to expose pet’s autodetect feature, such that the region of interest needs to be marked with #pragma first.)

4.3 PENCIL programs

Live-range reordering has been very instrumental in the experiments of Baghdadi, Beaugnon, et al. (2015). Almost all of these inputs have assignments to scalars inside the loop bodies. This means that without live-range reordering (or some other means of dealing with the scalar induced false dependences), the execution order would be completely serialized, leaving PPCG no option but to generate (unoptimized) CPU code for almost all the inputs, making an evaluation of the generated OpenCL code impossible. Note that these inputs were either written by hand or converted automatically from a DSL. In both cases, the use of temporary scalars was the most natural way of writing or generating these inputs. No attempt has been made to rewrite these inputs without the use of temporary scalars.

Take, for example, the image processing benchmark suite used by Baghdadi, Beaugnon, et al. (2015). Only one of these benchmarks does not have any assignments to scalar values in the loop bodies. This means that without live-range reordering support, PPCG would only be able to translate this single benchmark to OpenCL. Note that the benchmarks in this particular suite are fairly simple, consisting of a single perfectly nested loop nest each. This means that the tiling criterion of Baghdadi, Cohen, et al. (2013) could be applied directly to the input schedule. It also means that the scheduler could simply ignore the false dependences and still derive a valid schedule. This is illustrated to some extent in Table 1, where this benchmark suite is called “image”. The table shows that although some condition schedule constraints (i.e., order dependences) were violated, causing the corresponding condition constraints (i.e., live-ranges) to be forced to be local, all of these condition constraints were local already up to that point. The same holds for the examples from Listing 1 and Listing 3. Note that the table does not show that those live-ranges would also be local with respect to subsequently computed band members had they not been forced to be local.

Some other inputs are considerably more complicated. Consider, for example, the SpearDE (Lenormand and Edelin 2003) STAP benchmark used by Baghdadi, Beaugnon, et al. (2015, Section IV.D). This benchmark consists of several regions that need to be optimized. Some of these are very simple, not even requiring any groups of condition constraints to be marked local, but some others are more complicated, including one that requires a band that has already been partially computed to be reset. In this latter case, simply ignoring the false dependences would therefore clearly result in an incorrect schedule. It should also be noted that the performance improvements obtained on this benchmark are partly due to loop fusion. That is, simply taking the original schedule and detecting tilable (and parallelizable) loop nests would not achieve the same results.

5. CONCLUSIONS

An approach has been presented for dealing with memory reuse without serializing the successive uses of memory, but also without increasing memory requirements and without preventing loop distribution. The approach is generic and relatively simple. It is based on a relaxed permutability criterion, allowing live-ranges to be reordered by a schedule band as long as they are local to that band, and it works by forcing live-ranges to be local in a band whenever they may end up getting reordered by that band. The processing is entirely local to a band. In rare cases, a band may need to be recomputed, possibly even several times, but no other part of the schedule is affected. The approach has been publicly available for more than a year and has been thoroughly tested, but had not been described in detail before.

---

Table 1: Total number of band members computed, number of groups of condition constraints marked local, number of these groups that were not local already and the number of band resets performed

<table>
<thead>
<tr>
<th>Input</th>
<th>member</th>
<th>localized</th>
<th>non-local</th>
<th>reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing 1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Listing 3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>image</td>
<td>29</td>
<td>78</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>STAP</td>
<td>191</td>
<td>122</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Acknowledgements
This research was partly supported by the European FP7 project CARP id. 287767 and the COPCAMS ARTEMIS project. The authors would also like to thank Baghdadi, Beaunong, et al. (2015), especially those who performed the experiments, for their feedback.

References
Grosser, Tobias, Sven Verdoolaege, and Albert Cohen (2015). “Polyhedral AST generation is more than scanning polyhedra”. In: ACM Transactions on Programming Languages and Systems 37.4, 12:1–12:50. doi: 10.1145/2743016