HAL
archives-ouvertes.fr

PENCIL support in pet and PPCG

Sven Verdoolaege

▶ **To cite this version:**

Sven Verdoolaege. PENCIL support in pet and PPCG. [Technical Report] RT-0457, INRIA Paris-Rocquencourt; INRIA. 2015. <hal-01133962v2>

HAL Id: hal-01133962
https://hal.inria.fr/hal-01133962v2

Submitted on 16 May 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PENCIL support in pet and PPCG

Sven Verdoolaege

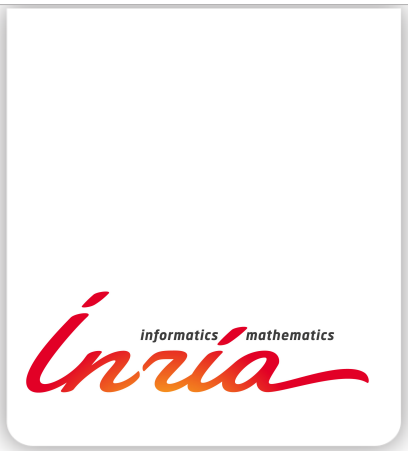
TECHNICAL REPORT

N° 457

March 2015

Project-Team PARKAS

ISSN 0249-0803 ISRN INRIA/RT--457--FR+ENG



PENCIL support in pet and PPCG

Sven Verdoolaege*

Project-Team PARKAS

Technical Report n° 457 — version 2 — initial version March 2015 — revised version May 2015 — 27 pages

Abstract: This report describes various changes to the polyhedral model extractor `pet` and the automatic parallelizer `PPCG`, including support for generic statements, arrays of structures, function summaries, dead code elimination and live-range reordering as well as support for annotations allowing the user to explicitly describe relations among program variables, kills and the absence of loop-carried dependences. Most of these changes are instrumental in supporting PENCIL in `PPCG`. The report also describes how synchronization is introduced by `PPCG`.

Key-words: polyhedral compilation, array of structures, function summary, dead code elimination, live-range reordering

Version 2 fixes the description of the contribution of Baghdadi et al. (2013a), describes the handling of live-ranges with shared sinks and reflects the recent changes in `PPCG` with respect to taking into account independences in the validity constraints.

* Also at KU Leuven

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Le support pour PENCIL dans `pet` et `PPCG`

Résumé : Ce rapport décrit plusieurs modifications apportées au logiciel permettant d'extraire un modéle polyhédrique `pet` et à l'outil de parallélisation automatique `PPCG`, y compris le support pour les instructions génériques, les tableaux de structures, la suppression du code mort et la réorganisation de l'ordre d'intervalles de vie, ainsi que le support pour la description explicite par l'utilisateur de relations entre variables et de l'absence de dépendances. La plupart de ces modifications contribuent aux support de PENCIL dans `PPCG`. Le rapport décrit aussi comment `PPCG` introduit la synchronisation.

Mots-clés : compilation polyhédrique, tableau de structures, sommaire de fonction, suppression du code mort, réorganisation de l'ordre d'intervalles de vie

1 Introduction

This document describes some of the changes that have been applied to `pet` (Verdoolaege and Grosser 2012) and `PPCG` (Verdoolaege et al. 2013b) since their initial descriptions, with a special focus on the changes that were needed to support PENCIL (Baghdadi et al. 2013b). Most of these changes have been available for some time already, but had never been described in detail before. The most recent changes have been made available in `ppcg-0.03-202-gc63f95d`. There are also some changes that are *not* covered by this document:

- OpenCL backend in `PPCG`

  This backend was initially contributed by Riyadh Baghdadi and has been further improved by Sven van Haastregt.

- support for `while`, `break` and `continue`

  The direct support for these constructs in `pet` is mainly useful for constructing weakly dynamic polyhedral process networks (Stefanov 2004) through run-time dependent dataflow analysis (Verdoolaege et al. 2013a). These constructs are not yet supported in `PPCG`. Instead, they are encapsulated inside generic statements (see Section 3.2).

- switch to schedule trees

  `pet` and `PPCG` now use the schedule trees of Verdoolaege et al. (2014) to represent schedules.

- multiple SCoPs

  `pet` now provides an interface for extracting multiple SCoPs from the same input file.

2 Terminology

An expression is called *static-affine* if it is a quasi-affine expression involving only symbolic constants and loop iterators.

The *statement instance set* is a set that contains an element for each dynamic execution of a statement in the analyzed program fragment.

3 Changes to `pet`

This section describes some of the changes that have been applied to `pet` since its initial description (Verdoolaege and Grosser 2012), with a special focus on the changes that were needed to support PENCIL (Baghdadi et al. 2013b).

3.1 Accesses

For each access in the analyzed fragment, the original version of `pet` (Verdoolaege and Grosser 2012) would only keep track of the following pieces of information.

- an `isl_map` representing the access relation

- a bit indicating whether this access is a read
- a bit indicating whether this access is a write

In order to support various extensions, the version of `pet` described by this document keeps track of the following pieces of information instead.

- an `isl_multi_pw_aff` representing the original index expression
- a reference identifier
- three `isl_union_map`s representing the may-reads, the may-writes and the must-writes
- a bit indicating whether this access is a read
- a bit indicating whether this access is a write
- a bit indicating whether this access is a kill

The reference identifier is needed to distinguish multiple accesses in the same statement. For example, support for live-range reordering (Baghdadi et al. 2013a) requires `PPCG` to keep track of which references cause any given dependence between statement instances. The reference identifiers can be used here to identify the references involved.

The index expression is mainly used for printing accesses in the transformed program. It is first rewritten and simplified in terms of the loop iterators of the transformed program and then optionally mapped to a shared or private memory tile (Verdoolaege et al. 2013b, Section 7). One advantage of an `isl_multi_pw_aff` representation is that the different index expressions can be processed individually. In particular, a disjunction appearing in one of them will not affect the others, whereas in an `isl_map` representation, the disjunction happens at the outer level such that it affects all indices. The main reason for introducing an explicit index expression, however, is that, as explained next, an `isl_map` no longer suffices to represent the access relations(s) and is therefore no longer available.

In the original version of `pet`, a slice of an array passed to a function was either completely read or completely written. With the support for function summaries (Section 3.6), `pet` can now construct a more accurate access relation by analyzing the function that is being called (or its summary function). This means that a distinction needs to be made between the elements that are read and those that are or may be written within a single array access. Furthermore, because the function may accept an array of structures (Section 3.3), the accessed elements may not all live in the same space. Instead of a single `isl_map`, three `isl_union_map`s are therefore required to express the access relations.

The kill bit is used to mark "accesses" that appear in kill statements (Section 3.8). These accesses do not actually access any data, but merely describe the set of elements that are being killed by the statement.

3.2 Generic statements

Whereas, originally, a statement in `pet` could only be an expression statement, it can now be any statement. In particular, the body of the statement is a

```
for (int x = 0; x < n; ++x) {
A:      s = f();
B:      while (P(x, s))
                s = g(s);
C:      h(s);
}
```

Listing 1: A program with a `while` loop

```
for (int i = 0; i < 100; ++i) {
        for (int j = 0; j < 100; ++j) {
                if (A[j]) {
                        x += 0;
                        continue;
                }
                x = 1;
        }
}
```

Listing 2: A program with a `continue`

`pet_tree` representing an AST. The support for such generic statements allows `pet` to (optionally) encapsulate any dynamic control inside its smallest containing statement. For example, the program in Listing 1 is considered to consist of three statements, with the second statement encapsulating the `while` loop and its body. The instance set is $\{ \mathtt{A}(x) : 0 \leq x < n; \mathtt{B}(x) : 0 \leq x < n; \mathtt{C}(x) : 0 \leq x < n \}$. The program in Listing 2 is considered to only have a single statement, encapsulating the inner `for` loop. This inner loop has dynamic control because of the `continue` statement governed by a condition that is not static-affine. The statement instance set for this program is $\{ S(i) : 0 \leq i < 100 \}$.

The generic statement AST in the body of a polyhedral statement is not further analyzed by `pet` but is instead considered as one indivisible entity. `pet` does take into account that there may be some dynamic control inside such statements and therefore treats all writes inside such a statement as a may-write. The dataflow analysis performed by `PPCG` is applied to these generic statements and their may-writes and it may therefore find dataflow from other statement instances to a given instance of a statement even if in reality the dataflow happens inside the statement. For example, the value of `t` read inside the `if`-statement in Listing 3 is clearly written inside the same instance of the `if`-statement. In practice, however, the result of the dataflow analysis will have the read depend on every previous iteration, as well as on the last write to `t` before the loop. This issue could be resolved by moving the dataflow analysis inside `pet` so that it could perform the dataflow analysis inside the generic statement before encapsulating it. As a work-around, the spurious dataflow can also be removed by inserting kills (Section 3.8).

```
for (int i = 0; i < 100; ++i) {
        if (A[i] > 0) {
                t = A[i];
                B[i] = t;
        }
}
```

Listing 3: Dataflow inside an encapsulating statement

```
struct s {
        struct {
                int a[10];
        } f[10];
};

void f()
{
        struct s s;

        for (int i = 0; i < 10; ++i)
                for (int j = 0; j < 10; ++j)
                        s.f[i].a[j] = i * j;
}
```

Listing 4: Accesses to structure fields

3.3 Structures

Accesses to fields of structures are represented using wrapped spaces. In particular, the data space is a wrapped map with the domain space referring to the accessed array and the range space identifying the field of the structure. The wrapped space is itself given a name that is composed of the array name and the field name. In case of nested field accesses, the wrapped space corresponding to the outer field access is taken as the domain of the wrapped space of the inner field access. For example, the nested field access in Listing 4 is represented as $\{ S(i,j) \to \texttt{s\_f\_a}(\texttt{s\_f}(\texttt{s}() \to \texttt{f}(i)) \to \texttt{a}(j)) \}$.

When collecting all may-reads, may-writes or must-writes in a `pet_scop`, `pet` returns access relations to the innermost fields. For example, for the program in Listing 5, the index expressions are `{ S[] -> a_b[a[] -> b[]] }`, `{ T[] -> a_a[a[] -> a[]] }`, `{ U[i] -> b[(i)] }` and `{ U[i] -> a[] }`. However, the collected read access relation is

```
{ U[i] -> a_b[a[] -> b[]]; U[i] -> a_a[a[] -> a[]] }      (1)
```

```
struct s {
        int a;
        int b;
};

int f()
{
        struct s a, b[10];

S:      a.b = 57;
T:      a.a = 42;
        for (int i = 0; i < 10; ++i)
U:              b[i] = a;
}
```

Listing 5: Accesses to structures

while the collected write access relation is

```
{
  S[] -> a_b[a[] -> b[]]; T[] -> a_a[a[] -> a[]];
  U[i] -> b_a[b[(i)] -> a[]];
  U[i] -> b_b[b[(i)] -> b[]]
}
```
(2)

3.4 Two-Phase Approach

The extraction of a polyhedral model now follows a two-phase approach. In the first phase, the `clang` AST is analyzed in a bottom-up fashion to find a region that is amenable for polyhedral analysis. At the same time, this part of the `clang` AST is also converted into an internal `pet_tree` representation. At this stage, `pet` has not yet detected any symbolic constants or outer loop iterators (because the tree is constructed bottom-up). All accesses are therefore represented as data dependent accesses (Verdoolaege and Grosser 2012, Section 4.1). For example, the read from `D` in Listing 6 is initially represented using the index expression

```
{ [[] -> [i0]] -> D[((i0) : i0 >= 0)] },
```
(3)

where `i0` represents the (currently) unknown result of the expression `i + m`. Inside the representation of this expression, the two arguments are represented as accesses to the variables `i` and `m` with index expressions `{ [] -> i[] }` and `{ [] -> m[] }`.

In the second phase, a `pet_scop` is extracted from the `pet_tree` constructed in the first phase. During a top-down traversal of this `pet_tree`, `pet` keeps track of a *context* containing known variable assignments and the domain for affine expressions. In particular, the domain collects the outer loop iterators, some of which may be virtual, along with their known bounds. Virtual iterators occur in infinite loops or loops with an unsigned loop iterator that may wrap. The

```
for (int i = 0; i < n; i++) {
        D[i] = D[i + m];
}
```

Listing 6: Simple loop

known variables assignments include symbolic constants, outer loop iterators and scalar variables that have been assigned a quasi-affine expression and that do not change within the current subtree. Each variable assignment maps an identifier to a piecewise quasi-affine expression. The domain of this expression is a prefix of the domain of the context. For example, when `pet` reaches the body of the loop in Listing 6, it has the following assignments:

```
i:  [i] -> [(i)]  [i] -> [(i)]
m:  [m] -> [] -> [(m)]                    (4)
n:  [n] -> [] -> [(n)]
```

Note that `m` and `n` are declared as parameters, while `i` is the first iterator in the context domain.

During the construction of a `pet_scop` from a `pet_tree`, each expression is *evaluated* in the context, meaning that the following steps are applied.

- Insert context domain.
- Plug in known values for read accesses.
- Try and plug in index expression arguments by converting them to affine expressions.
- Plug in function summaries. See Section 3.6.

Considering once more the loop body in Listing 6, inserting the context domain `{ [i] }` into the index expression of the read (3) results in the updated index expression

```
{ [[i] -> [i0]] -> D[((i0) : i0 >= 0)] }.          (5)
```

In the same way, the index expressions of the nested accesses are updated to `{ [i] -> i[] }` and `{ [i] -> m[] }`. The latter two index expressions access scalars for which a known value is available in the context, so they are further updated to `{ [i] -> [(i)] }` and `[m] -> { [i] -> [(m)] }`. At this point, `pet` tries to convert the `+` operation in the argument of the read of `D` into a quasi-affine expression. This operation succeeds here and results in the expression `[m] -> { [i] -> [(i + m)] }`. This result can then be plugged into the index expression (5) resulting in

```
[m] -> { [i] -> D[((i + m) : i + m >= 0)] }.       (6)
```

Note that as explained next, `pet` also takes into account signed integer overflow, so the actual result is slightly more complicated.

Next to the context in which expressions are evaluated and which is used during the construction of a `pet_scop`, a `pet_scop` itself also keeps track of a set called "context". This context collects constraints on the symbolic constants

```
void foo(int n, int m, int S,
         int D[const restrict static S])
{
         __pencil_assume(m > n);
         for (int i = 0; i < n; i++) {
                 D[i] = D[i + m];
         }
}
```

Listing 7: Explicit assumptions

that can be assumed to hold because the code represented by the `pet_scop` could not be executed otherwise. These constraints are derived from the bounds on the integer types used to declare the variables corresponding to the symbolic constants, the absence of signed integer overflow and the absence of negative array sizes and array indices. Note that while negative array indices are allowed in C99, they are not allowed by `pet` because they may lead to aliasing. During the construction of a `pet_scop`, the context of the `pet_scop` does not only refer to the symbolic constants, but also to the outer loop iterators. Each of these outer loop iterators is projected out on the way back up the `pet_tree`, from innermost to outermost. In particular, when leaving a loop with loop iterator j and with outer loop iterators $\mathbf{i}$, if $\mathrm{dom}(\mathbf{i}, j)$ represents the known constraints on the outer and current loop iterators and $\mathrm{valid}(\mathbf{i}, j)$ represents the current context inside the loop construct, then the context outside the loop is $\forall j : \mathrm{dom}(\mathbf{i}, j) \implies \mathrm{valid}(\mathbf{i}, j)$.

3.5 Explicit Assumptions

As explained in the previous section, `pet` derives necessary conditions on the symbolic constants from the bounds of the corresponding integer types and the absence of signed integer overflow and negative array sizes and array indices. The user is allowed to add additional constraints through a "call" to `__pencil_assume`. The argument of this call is a boolean condition that is guaranteed to hold at that position by the user. While this boolean condition may have any form, only those that are quasi-affine expressions in the symbolic constants and outer loop iterators are effectively taken into account.

Consider, for example, the program in Listing 7. Due to the explicit assumption, the final context satisfies the constraint $m > n$. In principle, this constraint could be used to prove that there are no loop-carried dependences in the program, but this context information is currently not exploited by `PPCG` during its dataflow analysis.

3.6 Function Summaries

Whenever `pet` comes across a function call during the first phase of the analysis, it checks if the body of the function is available. If so, a `pet_scop` is constructed for the entire function body. A failure to extract such a `pet_scop` is currently considered to be an error. For PENCIL input, this should not be a problem since the entire source file is supposed to conform to the PENCIL restrictions. Before

```
int f(int i);
int maybe();

struct s {
        int a;
};

void set_odd(int n, struct s A[static n])
{
        int t;

S:      t = f(0);
        for (int i = 1; i < n; i += 2)
                if (maybe())
T:                      A[i].a = t;
}

void foo(int n, struct s B[static 2 * n])
{
#pragma scop
        set_odd(2 * n, B);
#pragma endscop
}
```

Listing 8: Program with function call

extracting a `pet_scop`, `pet` first analyzes the arguments of the function and groups them into three classes

- integer arguments, i.e., those with an integer type,
- array arguments, i.e., those that may access data from the caller, and
- other arguments.

The `pet_scop` is then extracted with the integer arguments as the only symbolic constants. During this extraction, the autodetect option is turned off to ensure that the entire body is extracted. After the extraction, a function summary is constructed that collects summary information about the function. In particular, for each argument, the function summary keeps track of its type and some type-dependent information. For integer arguments, this extra piece of information is the identifier. For array arguments, three `isl_union_map`s are stored corresponding to the may-reads, the may-writes and the must-writes. The domains of these relations are formed by a space with a coordinate for each integer argument. The access relations are obtained by collecting the sets of accessed elements from the `pet_scop` extracted from the function body, distributing them over the array arguments and then replacing the symbolic constants by the corresponding local domain coordinates.

Consider, for example, the program in Listing 8. While analyzing the body of `foo`, `pet` runs across a call to `set_odd`. This function has two arguments,

`n` and `A`. The first is recognized as an integer argument, with identifier `n`, and the second argument is recognized as an array argument. The entire `set_odd` function body is therefore analyzed with `n` as the only symbolic constant. Assuming the option to encapsulate dynamic control is turned on, the body of the `for` loop is considered as a monolithic statement performing a may-write. The may-write access relation for the entire body is therefore

```
[n] -> { S[] -> t[];
         T[i] -> A_a[A[i] -> a[]] :
         exists (e0: 2e0 = -1 + i and
                 1 <= i <= -1 + n) }
```
(7)

The must-write access relation is

```
[n] -> { S[] -> t[] }
```
(8)

while the may-read access relation is

```
[n] -> { T[i] -> t[]:
         exists (e0: 2e0 = -1 + i and
                 1 <= i <= -1 + n) }
```
(9)

Projecting out the statement instances of the body (which are not relevant to the caller) from the may-write access relation, results in

```
[n] -> { t[]; A_a[A[i] -> a[]] :
         exists (e0: 2e0 = -1 + i and
                 1 <= i <= -1 + n) }
```
(10)

The array arguments are then considered in turn and for each of them the extent is determined in the context of the same symbolic constants. For the second argument, this extent is

```
[n] -> { A[i0] : i0 >= 0 and i0 <= -1 + n }.
```
(11)

Since the access relations are all given in terms of the innermost fields, this extent first needs to be mapped to those innermost fields as well, resulting in

```
[n] -> { A_a[A[i0] -> a[]] : i0 >= 0 and i0 <= -1 + n }.
```
(12)

Intersecting this extent with the may-write access (10) results in the may-writes for this array argument

```
[n] -> { A_a[A[i] -> a[]] :
         exists (e0: 2e0 = -1 + i and
                 1 <= i <= -1 + n) }
```
(13)

The symbolic constants in this set are then replaced by domain coordinates resulting in

```
{ [n] -> A_a[A[i] -> a[]] :
  exists (e0: 2e0 = -1 + i and 1 <= i <= -1 + n) }
```
(14)

The may-reads and the must-writes for this array argument are empty.

During the construction of a `pet_scop` from the `pet_tree` representing the body of `foo`, the call expression is evaluated in its context, which performs the steps detailed in Section 3.4 and at the end plugs in the corresponding function summary, if any. Plugging in a function summary involves the following steps. The integer arguments in the function call are converted to affine expressions in the current context, if possible. Otherwise, additional arguments are added to the context domain, resulting in data dependent expressions (Verdoolaege and Grosser 2012, Section 4.1). For each array argument in the function summary, if the corresponding argument in the function call is effectively an access expression, then integer arguments in the function call are plugged into the access relations in the function summary and combined with the index expression of the array access in the function call to produce access relations for this array access.

Returning to the example of Listing 8, there is only a single integer argument. The actual argument in the function call can in this case be represented as an affine expression in terms of the context domain

```
[n] -> { [] -> [(2n)] }.                                    (15)
```

Since there is only one integer argument, collecting all integer arguments results in the same affine expression here. Plugging in this parameter binding in the may-write access relation (14), results in

```
[n] -> { [] -> A_a[A[i] -> a[]] :
          exists (e0: 2e0 = -1 + i and                      (16)
                      1 <= i <= -1 + 2 * n) }
```

Finally, this access relation is combined with the index expression `{ [] -> B[] }`, resulting in the final may-write access relation

```
[n] -> { [] -> B_a[B[i] -> a[]] :
          exists (e0: 2e0 = -1 + i and                      (17)
                      1 <= i <= -1 + 2 * n) }
```

Combining the index expression of the caller with the access relations in the function summary involves dropping the name of the outer array in the access relation and pasting the index expressions of this outer array to those of the inner array of the index expression. In the example above, the dropped outer array name is `A` and its single index expression is added to the initially zero index expressions of `B`.

3.7 Summary Functions

In some cases, the access relations derived by `pet` from a called function may be too inaccurate. In the extreme case, no code may be available for the function such that `pet` can only assume that every element of the passed arrays is accessed. In order to obtain more accurate access relations, the user may tell `pet` to derive the access relations not from the actual function body, but from some other function with the same signature. Such a function is called a summary function and is treated in exactly the same way as in Section 3.6 for

```
int f(int i);
int maybe();

struct s {
        int a;
};

void set_odd_summary(int n, struct s A[static n])
{
        for (int i = 1; i < n; i += 2)
                if (maybe())
                        A[i].a = 0;
}

__attribute__((pencil_access(set_odd_summary)))
void set_odd(int n, struct s A[static n]);

void set_odd(int n, struct s A[static n])
{
        for (int i = 0; i < n; ++i)
                A[2 * f(i) + 1].a = i;
}

void foo(int n, struct s B[static 2 * n])
{
#pragma scop
        set_odd(2 * n, B);
#pragma endscop
}
```

Listing 9: Summary function

the purpose of deriving access relations of the calling statement. In particular, `pet` only collects the accesses in the function body and does not care about the order in which they are executed.

Listing 9 illustrates the use of summary functions. Since `pet` is unable to analyze the index expression in the body of the `set_odd` function, it would derive the may-write access relation

```
{ [n] -> A_a[A[i] -> a[]] : 1 <= i <= -1 + n }
```
(18)

for its function summary, missing the fact that only the odd elements could possibly be accessed. By writing a summary function `set_odd_summary` where this fact is more explicitly available and by marking it as a summary function for `set_odd` through the `pencil_access` attribute, the user can make `pet` derive the more accurate may-write access relation (14) of the previous section.

```
void foo(int n, int A[n][n], int B[n][n])
{
        for (int i = 0; i < n; ++i)
                #pragma pencil independent
                for (int j = 0; j < n; ++j)
                        B[i][A[i][j]] = i + j;
}
```

Listing 10: Explicitly marked absence of dependence

3.8 Kills

A kill statement in a `pet_scop` represents the fact that no dataflow on the killed data elements can pass through any instance of the statement. This information can be used during dataflow analysis to stop the search for potential sources on data elements killed by the statement.

Whenever `pet` comes across a variable declaration, two kill statements that kill the entire array are introduced, one at the location of the variable declaration and one at the end of the block that contains the variable declaration. Note that, currently, only single variable declarations are supported by `pet`. Any variable declared inside a statement encapsulating dynamic control is killed before and after the statement.

A user can also introduce explicit kills by adding a "call" to `__pencil_kill`. A kill statement is introduced for each argument passed to `__pencil_kill`. Section 4.4 describes an example of the use of `__pencil_kill` in Listing 12.

3.9 Absence of Dependences

Whenever there is control or an index expression that is not static-affine, the may-read and may-write access relations may be strict overapproximations of the actual accesses. These may therefore result in dependences being detected that do not occur in practice. The user can exclude such dependences by introducing a `#pragma pencil independent`. When placed in front of a loop, this pragma means that, assuming that a variable that is declared inside the loop is considered private to any given iteration, the iterations of the loop may be freely reordered with respect to each other, including reorderings that result in (partial) overlaps of distinct iterations. In particular, the user asserts through this pragma that no dependences need to be introduced to prevent such reorderings. For example, in the program of Listing 10, the user may have additional information about the `A` array ensuring that each row has all distinct entries. This means that each iteration of the inner loop accesses a different element of the `B` array such that the iterations of this loop may be freely reordered.

Since `pet` itself does not perform any form of dataflow analysis, it needs to propagate the independence information to its users. In particular, for each `#pragma pencil independent`, it keeps track of all pairs of statement instances for which no dependence needs to be introduced as well as all variables that are local to the body of the loop. This information is currently only collected for `for`-loops with static-affine loop bounds and a constant increment. The representation is fairly inefficient because it is quadratic in the number of statements

inside the loop body. The support for these pragmas predates the introduction of schedule trees. Now that `pet` keeps track of schedule trees, it should be possible to come up with a more efficient representation. For example, a mark node could be introduced on top of the band node corresponding to the loop decorated with the pragma. Ultimately, however, it may be best for `pet` to perform dataflow analysis itself such that this information no longer needs to be propagated to its users.

4 Changes to `PPCG`

This section describes some of the changes that have been applied to `PPCG` since its initial description (Verdoolaege et al. 2013b), with a special focus on the changes that were needed to support PENCIL (Baghdadi et al. 2013b).

4.1 Dead Code Elimination

Dead code elimination is performed by constructing the set of statement instances that may need to be executed. Initially, this is the set of statement instances that may be the last to write to an array element. In particular, a dataflow analysis is performed where the "sinks" are the must-writes and the kills, while the "potential sources" are the may-writes. All the may-writes that appear as the source in the result of this dataflow analysis are definitely killed by the corresponding must-write or kill and therefore cannot possibly be the last to write to the corresponding array element. The statement instances performing the remaining set may-writes then form the initial set of statement instances that need to be executed. This set is extended with all the statement instances that perform a function call since `pet` currently does keep track of "pure" functions and so `PPCG` can only assume that each function call needs to be performed.

The statement instances in the initial set may depend on other statement instances through potential flow dependences, which therefore need to be executed as well. The dataflow dependence relation is therefore applied to the current set and the result is added to this set. This process then needs to be repeated until it stops producing additional elements. However, a naive application of this scheme could lead to an infinite loop in case of parametric bounds on the original instance set since each iteration of the scheme would consider ever increasing values of the symbolic constants. In order to avoid this trap, a form of widening (Cousot and Cousot 1992) is applied. In particular, after each addition of new statement instances, the current set is replaced by its integer affine hull (Verdoolaege 2010) and then intersected with the complete set of statement instances. Each iteration therefore increases the dimension of the instance set of at least one statement. Since the number of statements is finite and since the dimension of each of the corresponding instance sets is finite, this process is guaranteed to terminate.

4.2 Structures

Support in `PPCG` is fairly basic at the moment. The fields of a structure are assumed not to alias with each other (or indeed anything else), so that dataflow

analysis does not need to be adjusted to support structures. If some element of a structure is accessed on the device, then the entire outer array containing the structure is copied as a whole to the device and back from the device (if needed). Arrays of structures are currently not considered for mapping to shared or private memory.

4.3 Kills

The kills introduced in Section 3.8 are used by `PPCG` in three steps,

- the dataflow computation,
- the computation of the set of live-out accesses, and
- the computation of the may-persist set.

The first two are explained in Section 4.4. The third is explained here.

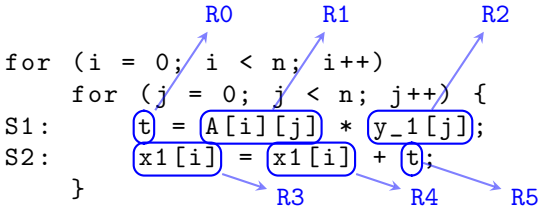
The may-persist set is used during the computation of which elements to copy in and/or out of the device. `PPCG` first computes the set of data elements that need to be copied out. This set is computed from the may-writes that occur within the kernel(s) for which the copying needs to be performed. Some of these may-writes may however only be needed within the given subtree of the schedule tree (which is entirely mapped to the device) and therefore may not need to be copied out. In particular, `PPCG` removes those may-writes that are only involved in local dataflow and that are not live-out. In the end, the copy-out set is approximated by the entire arrays for which at least one element needs to be copied out. The copy-in set is similarly derived from the may-reads. However, there may be elements that are copied out that are not necessarily written inside the relevant part of the schedule tree. That is, there may be may-writes that are not also must-writes or there may be elements in the approximation of the copy-out set that may not be possibly written at all. In order to preserve their values with respect to the host, these arrays therefore need to be added to the copy-in set. This is where the may-persist set comes in. The host code may not care about some of these elements and the copy-in would be a waste of time and energy. Instead of adding all elements that are copied out but that may not have been written to the copy-in set, `PPCG` only adds those that are also elements of the may-persist set or that may be involved in dataflow that crosses the schedule subtree at hand.

The may-persist set then is computed from the extent of all arrays that are not locally declared inside the program fragment, mapped to the innermost fields. In particular, the may-persist set consists of all elements of those arrays except those elements that are definitely written or killed somewhere in the program fragment.

Note that the way the copy-in set is updated to include those elements of the copy-out set that may not have been written is similar to the way approximations are handled by Alias et al. 2011.

4.4 Live-range Reordering

Baghdadi et al. (2013a) describe a relaxed permutability criterion that can be used to decide whether a schedule band can be tiled. However, the criterion



Listing 11: One loop from the *mvt* kernel, with reference identifiers

is only used *after* the schedule band has already been constructed. In the implementation on top of `Pluto` (Bondhugula et al. 2008) described by Baghdadi et al. (2013a), this is achieved by reinterpreting the bands computed by `Pluto`. The same criterion can however also be used *during* the construction of schedule bands and this is how it is implemented in `PPCG`.

The main idea of Baghdadi et al. (2013a) is that dependences that are only introduced to prevent live-ranges from overlapping may be ignored during tiling if the corresponding live-ranges are local to the schedule band that is being tiled. That is, if the source and the sink of a live-range are mapped to the same iteration of the schedule band, then this live-range may be freely reordered with respect to other live-ranges on the same array elements by the schedule band and therefore does not prevent tiling. `PPCG` exploits this idea during the very construction of permutable bands.

Since a dependence between two statement instances may be caused by several pairs of accesses inside those two statements, especially if these are generic statements encapsulating dynamic control (Section 3.2), it is important to keep track of the accesses at hand. `PPCG` therefore computes so-called "tagged" dependence relations, where domain and range do not simply refer to a statement instance, but rather to a pair of statement instance and reference identifier. Consider, for example, the program fragment of Baghdadi et al. (2013a, Fig. 7), reproduced in Listing 11 with reference identifier annotations. The tagged dataflow relation of this fragment is

```
[n] -> { [S1[i,j] -> R0[]] -> [S2[i,j] -> R5[]] :
         0 <= i, j < n;
         [S2][i,j] -> R3[]] -> [S2[i,j+1] -> R4[]] :
         0 <= i < n and 0 <= j < n - 1 }
```
(19)

While constructing a schedule band, the scheduler needs to make sure that either the source and the sink of a live-range are assigned the same value by the band or that some validity constraints are respected that ensure that the live-range does not overlap with any other live-range on the same data element. The mechanism used by `isl` for communicating such information is the *conditional validity* schedule constraint. Such a constraint consists of two parts, a condition and a (conditional) validity constraint. Both of these are tagged relations, i.e., they relate two pairs of statement instance and reference identifier. The scheduler ensures that for each element of the condition relation, if domain and range are scheduled apart, that then all *adjacent* conditional validity constraints are satisfied. Given an element $\{ (S(\mathbf{i}) \to U()) \to (T(\mathbf{j}) \to V()) \}$ of the condition relation, an adjacent conditional validity constraint is one where either the domain is the element $\{ (T(\mathbf{j}) \to V()) \}$ or the range is the element

$\{ (S(\mathbf{i}) \to U()) \}$.

In order to exploit this mechanism, `PPCG` computes the following dependences. First recall that the dataflow analysis of `isl` takes as input a schedule, a set of "sink" accesses, a set of "must-source" accesses and a set of "may-source" accesses. The resulting dataflow relation matches each sink access to the last must-source access (according to the schedule) that accesses the same data element (if any) as well as to any intermediate may-source that accesses the same data element. If there is no matching must-source, then the sink is matched to all previous may-sources.

- the *flow* dependences are computed using the `isl` dataflow analysis with the may-reads as sinks, the may-writes as may may-sources and as must-sources the union of the must-writes and the kills (Section 3.8). The kills are subsequently removed from the domain of the dataflow relation to only retain dataflow between actual writes and reads. Removing these kills is important for the use of the dataflow relation for dead code elimination (Section 4.1), as it should not add kill statement instances to the set of statement instances that need to be executed.

- the *order* dependences are the ones that need to ensure that live-ranges (i.e., dataflow dependences) do not overlap in case they are not local to a schedule band. In particular, the order dependences maintain the initial order of the live-ranges and relate any read or unmatched write to any later write. An unmatched write is one that does not occur as the source in any live-range. In principle, such writes should have been removed by dead code elimination (Section 4.1), but the write may occur in a statement that performs a call or it may have been introduced due to the widening step. It is important to include such unmatched writes in the order dependences to prevent these writes from getting scheduled in the middle of a later live-range. The writes that do appear in a live-range do not need to be included since they are necessarily executed before the corresponding read(s) and these reads *are* prevented from getting scheduled in the middle of a later live-range.

- the *forced* dependences are validity dependences that need to be enforced even when live-range reordering is used and consist of two parts, the *external* dependences and the order dependences between sources with a shared sink. The *external* dependences ensure that a live-in read remains live-in and that a live-out write remains live-out. They are computed by relating any live-in read to any later may-write to the same data element and similarly any may-write to any later live-out may-write to the same data element. The order dependences between sources with a shared sink ensure that such sources retain their relative execution order. If there are any may-writes, then there may be several flow dependences with the same sink. Only one of the corresponding sources will be the actual source at run-time, but some of the earlier potential sources may also be executed and they need to be prevented from getting executed *after* the actual source in the transformed program. Dependences are therefore introduced between any source of a live-range and any later source with a shared sink.

- the *false* dependences are kept for compatibility with the mode of `PPCG` where live-ranges are not reordered. They are computed using the `isl` dataflow analysis with the may-writes as sinks, the union of the may-writes and the may-reads as may-sources and the must-writes as must-sources.

Using these dependence relations, the schedule constraints for the `isl` scheduler are set 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 and the false dependences. Again, the reason for adding the false dependences is compatibility with the mode of `PPCG` where live-ranges are not reordered.
- the *coincidence* constraints, i.e., the constraints that determine which members of a band are considered "coincident", are set to the union of the flow dependences, the forced dependences and those order dependences that are derived from non-scalars. The reason for adding these order dependences is that `PPCG` currently only supports privatization of scalars. If there is an order dependence between two iterations of a schedule dimension then this means that the two iterations may access the same data elements. If this schedule dimension is considered to be coincident, then this data element needs to be privatized. Since `PPCG` can only handle such privatization for scalars, the presence of an order constraint between non-scalars should therefore prevent the schedule dimension from being considered coincident.
- the *conditional validity* constraints have the flow dependences as condition and (all) order dependences as (conditional) validity constraints.

The order dependences on the scalars are used to check which scalars should forcibly be mapped to private memory.

Despite the fact that tiling was the main motivation for Baghdadi et al. (2013a), it should be noted that live-range reordering as implemented in `PPCG` is also useful for other transformations. Consider, for example, the example program of Mehta (2014, Figure 5.4 (a)), reproduced in Listing 12, with a couple of minor changes. First, the lower bound on the `k1`-loop has been changed from `0` to `1` to avoid an out-of-bounds access in statement `S2`. Second, an explicit kill has been added to tell `PPCG` that the contents of `a0` and `am1` do not need to be preserved. Otherwise, `PPCG` would have to make sure that the final writes to these scalars remain the last writes, implying that a loop containing such a final write could not be considered to be coincident. The approach of Mehta (2014, Section 5.4) appears to be more restrictive than live-range reordering. In particular, it only allows the outer two loops in Listing 12 to be fused, while live-range reordering allows all three loops to be fused.

Without live-range reordering, the dependences on `a0` prevent essentially any reordering and the computed schedule tree is shown in Figure 1 (with the root domain node omitted). The generated C code would be essentially the same as the input code. With live-range reordering, the loops can be fused as long as no live-ranges overlap. In this case, the schedule tree in Figure 2 is computed (again with the root domain node omitted). Without the kills on `a0` and `am1`, none of

```
void foo(int Nx, int Ny, int Nz, int a[Nx][Ny][Nz],
         int x[Nx][Ny][Nz], int rho[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]);
            }
        }
    }

    __pencil_kill(a0, am1);
}
```

Listing 12: Example program from Mehta (2014, Figure 5.4 (a))

```
schedule: "[Nz, Ny, Nx] -> [{
        S5[i2, j2, k2] -> [(Nx + i2)];
        S1[i1, j1, k1] -> [(i1)];
        S2[i1, j1, k1] -> [(i1)];
        S3[i1, j1, k1] -> [(i1)];
        S4[i2, j2, k2] -> [(Nx + i2)] }]"
permutable: 1
child:
  set:
  - filter: "[Nz, Ny, Nx] -> { S2[i1, j1, k1];
                S3[i1, j1, k1]; S1[i1, j1, k1] }"
    child:
      schedule: "[Nz, Ny, Nx] -> [{
                S1[i1, j1, k1] -> [(j1)];
                S2[i1, j1, k1] -> [(j1)];
                S3[i1, j1, k1] -> [(j1)] }]"
      permutable: 1
      child:
        schedule: "[Nz, Ny, Nx] -> [{
                S1[i1, j1, k1] -> [(k1)];
                S2[i1, j1, k1] -> [(k1)];
                S3[i1, j1, k1] -> [(k1)] }]"
        permutable: 1
        child:
          sequence:
          - filter: "[Nz, Ny, Nx] -> { S1[i1, j1, k1] }"
          - filter: "[Nz, Ny, Nx] -> { S2[i1, j1, k1] }"
          - filter: "[Nz, Ny, Nx] -> { S3[i1, j1, k1] }"
  - filter: "[Nz, Ny, Nx] -> { S4[i2, j2, k2];
                               S5[i2, j2, k2] }"
    child:
      schedule: "[Nz, Ny, Nx] -> [{
                S5[i2, j2, k2] -> [(j2)];
                S4[i2, j2, k2] -> [(j2)] }]"
      permutable: 1
      child:
        schedule: "[Nz, Ny, Nx] -> [{
                S5[i2, j2, k2] -> [(k2)];
                S4[i2, j2, k2] -> [(k2)] }]"
        permutable: 1
        child:
          sequence:
          - filter: "[Nz, Ny, Nx] -> { S4[i2, j2, k2] }"
          - filter: "[Nz, Ny, Nx] -> { S5[i2, j2, k2] }"
```

Figure 1: Schedule tree for Listing 12 without live-range reordering

```
schedule: "[Nz, Ny, Nx] -> [
        { S5[i2, j2, k2] -> [(i2)];
          S1[i1, j1, k1] -> [(i1)];
          S2[i1, j1, k1] -> [(i1)];
          S3[i1, j1, k1] -> [(i1)];
          S4[i2, j2, k2] -> [(i2)] },
        { S5[i2, j2, k2] -> [(j2)];
          S1[i1, j1, k1] -> [(j1)];
          S2[i1, j1, k1] -> [(j1)];
          S3[i1, j1, k1] -> [(j1)];
          S4[i2, j2, k2] -> [(j2)] },
        { S5[i2, j2, k2] -> [(1 + k2)];
          S1[i1, j1, k1] -> [(k1)];
          S2[i1, j1, k1] -> [(k1)];
          S3[i1, j1, k1] -> [(k1)];
          S4[i2, j2, k2] -> [(1 + k2)] }]"
permutable: 1
coincident: [ 1, 1, 0 ]
child:
  sequence:
  - filter: "[Nz, Ny, Nx] -> { S1[i1, j1, k1] }"
  - filter: "[Nz, Ny, Nx] -> { S2[i1, j1, k1] }"
  - filter: "[Nz, Ny, Nx] -> { S3[i1, j1, k1] }"
  - filter: "[Nz, Ny, Nx] -> { S4[i2, j2, k2] }"
  - filter: "[Nz, Ny, Nx] -> { S5[i2, j2, k2] }"
```

Figure 2: Schedule tree for Listing 12 with live-range reordering

```
for (int c0 = 0; c0 < Nx; c0 += 1)
  for (int c1 = 0; c1 < Ny; c1 += 1)
    for (int c2 = 1; c2 < Nz; c2 += 1) {
      a0 = a[c0][c1][c2];
      am1 = a[c0][c1][c2 - 1];
      x[c0][c1][c2] = (a0 + am1);
      a0 = a[c0][c1][c2 - 1];
      rho[c0][c1][c2 - 1] =
            (a0 + (x[c0][c1][c2] - x[c0][c1][c2 - 1]));
    }
```

Listing 13: Plain C code generated from the schedule tree in Figure 2

the schedule dimensions in the band would be considered coincident. Since `PPCG` insists that the outer schedule dimension is coincident (when generating CUDA or OpenCL code), this schedule would be rejected without the kills. Generating plain C code for the schedule in Figure 2 produces the code in Listing 13.

4.5 Absence of Dependences

The information collected by `pet` from a `#pragma pencil independent` as explained in Section 3.9 is currently only exploited by `PPCG` when live-range reordering (Section 4.4) is enabled (which is the default). The reason is that the live-range reordering support already takes care of ensuring that the live-ranges of variables local to the marked loops either do not overlap or that they are mapped to private variables on the device.

Currently, the information about which pairs of statement instances do not need to be protected from reordering or overlapping is only used *after* the dependences have been computed. In particular, the effect on the schedule constraints is as follows:

- the independent pairs of statement instances of which the first is *not* a must-write are removed from the flow dependences. Those pairs where the first is a must-write cannot simply be removed because this must-write may have killed other dependences that would have to be reinstated if the dependence were removed.

  Note that it would arguably be better to adjust the dataflow analysis such that the unwanted dependences would not be added in the first place rather than trying to remove them afterwards. Such an adjusted dataflow analysis would be similar to the one proposed by Collard and Griebl (1996).

- the independent pairs of statement instances are removed from the external dependences. Note that the order dependences between sources with a shared sink are computed from the flow dependences after the independent pairs of statement instances have been removed from these flow dependences. These dependences therefore do not require any further adjustment.

- the independent pairs of statement instances are removed from the order dependences on arrays that are not local to the corresponding loop. Those that *are* local to the corresponding loop still need to be protected from overlapping, but for the non-local arrays, the user has guaranteed that no such protection is required.
- the independent pairs of statement instances are removed from the flow and forced dependences for the purpose of computing the *coincidence* constraints. Recall that the coincidence constraints are formed by taking the union of the flow dependence, the forced dependence and the order dependences on non-scalars. The independent pairs of statement instances have already been removed from the order dependences on non-local arrays. In the end, this means that the independent pairs of statement instances are removed from all contributors to the coincidence constraints except the order dependences on the local arrays.

4.6 Synchronization

There are two reasons why synchronization may need to be introduced.

- If any data is copied to/from shared memory, then the thread that copies the data may be different from the thread that reads/writes the data. Synchronization therefore needs to be added when any such copying is being performed. In particular, if there is any copying from global memory to shared memory (before the core computation), then synchronization needs to be added between this copying (which writes to shared memory) and the core computation (which reads from shared memory) and also between the core computation and the next iteration of the copying. Similarly, if there is any copying from shared memory back to global memory (after the core computation), then synchronization needs to be added between the core computation (which writes to shared memory) and the copying (which reads from shared memory) and also between this copying and the next iteration of the core computation.
- If any data is written to global memory, then it may be read again by the same block in a later iteration and this subsequent read may be performed by a different thread. Synchronization therefore needs to be added after any write to global memory that may be read again by the same block (and within the same iteration of all outer bands).

Note that `PPCG` currently does not treat these different types of synchronization differently in any way. In particular, they are not mapped to different kinds of barriers when targeting OpenCL.

Let us now consider in more detail how `PPCG` adds synchronization. `PPCG` first collects all the write references that require any synchronization on global memory. These are write references for which one of the corresponding reads or a conflicting write may occur in the same block but in a different thread. They are computed from the potential flow dependences and the pairs of potential writes accessing the same memory location that have the same schedule prefix for the kernel launch, but a different schedule prefix for the node that introduces the thread identifiers. If any of these references that require synchronization is mapped to global memory, then synchronization is added after

the core computation unless the schedule depth of the node that is mapped to threads is the same as the schedule depth of the kernel. In the case where these depths are the same, there can be no loops within the kernel but outside of the synchronization, meaning that there is nothing to synchronize against.

Whenever `PPCG` adds extra statements for copying to/from shared or private memory, it also checks if any further synchronization is needed. Such statements and potential synchronization are added for each reference group separately. The reference groups are currently considered in a fairly arbitrary order. It may be better to first consider those groups that need to have copy statements introduced deeper in the schedule tree such that that the synchronization introduced for these groups could be taken into account when considering synchronization at higher levels.

If the reference group is mapped to private memory, then synchronization is only added when data is copied back from private memory to global memory. There is no need to add synchronization between the core computation and the copy because the copy is necessarily performed by the same thread. There is also no synchronization required before any copying from global memory to private memory because synchronization is introduced after every write to global memory that may have a corresponding read in the same kernel. Furthermore, synchronization is only added if the reference group contains any of the writes that require synchronization and if the schedule depth where the copy statement is introduced is strictly greater than the schedule depth of the kernel.

If the reference group is mapped to shared memory, then synchronization may be needed both after the copying and after the core computation. In case of a read from global memory to shared memory, this means that synchronization may be added before and after the core computation. Before adding this synchronization, `PPCG` first checks if such synchronization has already been introduced at the schedule depth where the copying is being performed or any deeper level. If so, then no additional synchronization needs to be introduced. Furthermore, the synchronization after the core computation is only added if the copy depth is strictly greater than the kernel depth. In case of a write from shared memory back to global memory, the synchronization is usually inserted before and after the copying. The synchronization before the copying comes after the core computation and can then also be avoided if such synchronization has already been introduced at the same or a deeper level. The synchronization after the copying is only introduced if the copy depth is strictly greater than the kernel depth. If the reference group contains any of the writes that require synchronization, then the synchronization is added after the copying. Otherwise, no synchronization is needed to protect the write to global memory, but the read from shared memory still needs to be protected from being overwritten by a later write. Note that the only later writes of concern are those performed by the core computation since any potential later read from global memory to shared memory is performed by the same thread that is performing the write from shared memory to global memory. It is therefore sufficient to add synchronization before the core computation at the schedule depth where the copying is introduced. As usual, this synchronization can be skipped if synchronization has already been introduced at the same or a deeper level.

Acknowledgements

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

References

[1] C. Alias, A. Darte, and A. Plesco. *Kernel offloading with optimized remote accesses*. Rapport de recherche RR-7697. INRIA, July 2011. [16]

[2] R. Baghdadi, A. Cohen, S. Verdoolaege, and K. Trifunovic. "Improved loop tiling based on the removal of spurious false dependences". In: *TACO* 9.4 (2013), p. 52. DOI: 10.1145/2400682.2400711. [1, 4, 16, 17, 19]

[3] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, et al. *PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs*. 2013. [3, 15]

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. DOI: 10.1145/1375581.1375595. [17]

[5] J.-F. Collard and M. Griebl. "Array Dataflow Analysis for Explicitly Parallel Programs". In: *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*. Euro-Par '96. London, UK, UK: Springer-Verlag, 1996, pp. 406–413. DOI: 10.1142/S0129626497000140. [23]

[6] P. Cousot and R. Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In: *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*. Ed. by M. Bruynooghe and M. Wirsing. Leuven, Belgium: LNCS 631, Springer-Verlag, 1992, pp. 269–295. DOI: 10.1007/3-540-55844-6_142. [15]

[7] S. Mehta. "Scalable Compiler Optimizations for Improving the Memory System Performance in Multi-and Many-core Processors". PhD thesis. University of Minnesota, 2014. [19, 20]

[8] T. Stefanov. "Converting Weakly Dynamic Programs to Equivalent Process Network Specifications". PhD thesis. Leiden University, Leiden, The Netherlands, Sept. 2004. [3]

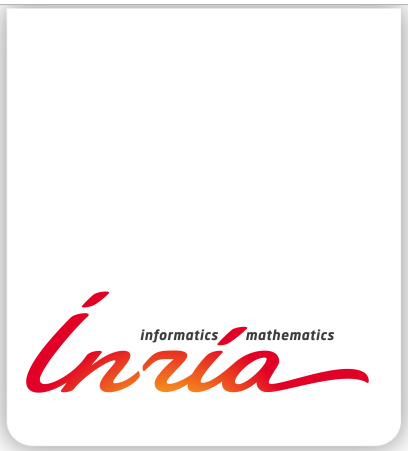
[9] S. Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49. [15]

[10] S. Verdoolaege and T. Grosser. "Polyhedral Extraction Tool". In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France, Jan. 2012. DOI: 10.13140/RG.2.1.4213.4562. [3, 7, 12]

[11] S. Verdoolaege, H. Nikolov, and T. Stefanov. "On Demand Parametric Array Dataflow Analysis". In: *Third International Workshop on Polyhedral Compilation Techniques (IMPACT'13)*. Berlin, Germany, Jan. 2013. DOI: 10.13140/RG.2.1.4737.7441. [3]

[12] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, et al. "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (2013), p. 54. DOI: 10.1145/2400682.2400713. [3, 4, 15]

[13] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria, Jan. 2014. DOI: 10.13140/RG.2.1.4475.6001. [3]



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803