Polyhedral Extraction Tool

Conference Paper · January 2012

DOI: 10.13140/RG.2.1.4213.4562

2 authors, including:

Sven Verdoolaege
National Institute for Research in Computer Science and Control
63 PUBLICATIONS **937** CITATIONS

SEE PROFILE

Polyhedral Extraction Tool

Sven Verdoolaege
Consultant for LIACS, Leiden
sverdool@liacs.nl

Tobias Grosser
INRIA/ENS, Paris
tobias.grosser@inria.fr

ABSTRACT

We present a new library for extracting a polyhedral model from C source. The library is based on clang, the LLVM C frontend, and isl, a library for manipulating quasi-affine sets and relations. The use of clang for parsing the C code brings advanced diagnostics and full support for C99. The use of isl allows for an easy construction and a powerful and compact representation of the polyhedral model. Besides allowing arbitrary piecewise quasi-affine index expressions and conditions, the library also supports some data dependent constructs and has special treatment for unsigned integers. The library has been successfully used to obtain polyhedral models for use in an equivalence checker, a tool for constructing polyhedral process networks, a parallelizer targeting GPUs and an interactive polyhedral environment.

1. INTRODUCTION AND MOTIVATION

The polyhedral model has been successfully used for several decades and is gradually finding its way into industrial compilers such as the IBM-XL [10] compiler, GCC (through graphite [24]) and LLVM/clang (through Polly [17]). All these compilers extract the polyhedral model from a low to medium level intermediate representation (IR). This makes the extraction independent of the input programming language and allows them to leverage compiler internal canonicalization and analysis passes to increase the amount of code they can analyze. However, when analyzing a compiler internal IR, relating the analysis to the input program becomes difficult. Hence, direct user feedback about unsupported constructs is normally not available. Also, the generated code is in most cases again a compiler internal IR. Synthesizing a higher level language is hard and even if successful, relating the resulting high level code to the original program would remain difficult. We conclude that analyzing a compiler internal IR may not be the best approach if user feedback is required or a high level language should be generated.

For some applications, including polyhedral source-to-source compilers such as Pluto [11] and PoCC[1], it is therefore more convenient to extract a polyhedral model from the source language, typically C or Fortran, or, more commonly, a subset and/or mixture of these languages. The parsers for these tools usually only analyze those program fragments, called static control parts (scops), that need to be converted to the polyhedral model. This means that information from outside of these scops, such as the types of variables or the sizes of arrays, are not available in the output or has to be redeclared using special annotations. By far the most popular such parser is `clan` [7], which is used by both Pluto and PoCC.

With our `pet` (`http://freecode.com/projects/libpet`), we have chosen a third option. We use a real C compiler to parse the source code and extract the polyhedral model directly from the high-level abstract syntax tree (AST). Although similar parsers have been developed in the past or are still in development, we believe that `pet` is the first to be publicly available. Furthermore, due to the use of a modern integer set library, it can, compared to other extractors, significantly relax the definition of a scop. The use of a real C parser, in our case clang, has many advantages. In particular, clang has full support for C99 [18], including variable length arrays (VLAs). This support is crucial for properly parsing the dynamically sized arrays versions of the PolyBench 3.1 benchmarks.[2] Additionally, clang generates very nice diagnostics, allowing us to clearly communicate to the user which constructs (if any) in the input code are (currently) not supported by `pet`.

Besides the above generic motivation, we were also faced with some specific needs for a better parser. The aim of the project that partially funded this work is to extend the `pn` tool [32] for constructing polyhedral process network to handle some dynamic constructs [23, 27]. The original polyhedral parser used by `pn`, called `pers`, was built on top of SUIF [2], which is no longer being actively maintained and has no support for C99. Furthermore, the construction of the polyhedral model was performed using a combination of some SUIF passes, the Omega library [20] and an ad-hoc constraints based internal representation. Extending `pers` was therefore not considered to be a viable option.

Similarly, our work on `PPCG`, a tool for generating GPU code

IMPACT 2012
Second International Workshop on Polyhedral Compilation Techniques
Jan 23, 2012, Paris, France
In conjunction with HiPEAC 2012.

http://impact.gforge.inria.fr/impact2012

[1] `http://pocc.sourceforge.net`
[2] `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`

from a sequential program using the polyhedral model, requires the sizes of the accessed arrays to generate the needed memory transfers. This information was not available in the output of `clan`, which was the extractor originally used by `PPCG`. Since `clan` does not even look at the array declarations, extending it to provide this information seemed non-trivial at best.

The desire to replace these two parsers by a single polyhedral parser lead to the requirement that `pet` should support the features of both tools. In particular, `pers` was also being used by an equivalence checker [31], which has been used to compare the outputs of different versions of `CLooG` [6] and which has also been extended to handle some data-dependent constructs [33]. `pet` was therefore designed to be able to parse both the output of `CLooG` and these data-dependence constructs.

There are several libraries available that can be used to manipulate a polyhedral model, including the Omega library [20], `PolyLib` [21], `PPL` [4] and `isl` [29]. Some polyhedral parsers prefer to use an internal and/or output representation that is independent of any polyhedral library. We believe however that the internal use of a powerful library such as `isl` significantly facilitates the development of a polyhedral parser. In fact, by leveraging the power of both `clang` and `isl`, the initial usable version of `pet` satisfying all the initial requirements and including most of the features described in this paper was written within a single man-month. Since both initial users of `pet` also use `isl`, the `isl` representation is also used in the output. Any other representation would only risk losing information. Finally, the use of the `isl` representation also allowed us to easily integrate `pet` into the interactive environment `iscc` [30].

2. OVERVIEW

The input to `pet` is valid C source code. A polyhedral model will be constructed for a fragment of this C code. In its default mode of operation, the fragment to be analyzed needs to be delimited by pragmas, in particular `#pragma scop` and `#pragma endscop`. If any construct inside this fragment cannot be analyzed by `pet`, then a warning is produced pointing out the unsupported construct. If the user specifies the `--autodetect` option then `pet` will try to detect a fragment that fits inside the polyhedral model. In this case, no warnings will be produced by `pet` as any code containing unsupported constructs will be considered to lie outside of the extracted code fragment. Supported constructs include expression statements, `if` conditions (see Section 3.2) and `for` loops (see Section 3.3). All index expressions are required to be piecewise quasi-affine (see Section 3.1), but some data dependent constructs are allowed as well (see Section 4.1.

The output is called a "scop" and consists of a context, a list of arrays and a list of statements. The context is a parameter set containing those parameters values for which the scop can be executed. Currently, the context is mainly used to ensure that all arrays have non-negative sizes. Following C99 6.7.5.2 `pet` can assume that a scop will only be executed with parameters that yield array sizes larger than zero. However, as both clang and gcc allow zero sized arrays in non pedantic mode, we also permit parameter values yielding arrays of size zero. For each array, we keep track of its "extent", its

```
void foo(int N)
{
        int a[N + 1];
#pragma scop
        for (int i = 0; i <= N; ++i)
U:              a[i] += i;
#pragma endscop
}
```

Figure 1: A trivial program

element type and (optionally) the set of possible values. The extent is a set defined in a space named after the array and containing all allowed array indices. In other words, the extent describes the size of the array. The set of possible values of an array may be specified by the user through a `#pragma value_bounds`.

Each statement consists of a line number, an iteration domain, (part of) a schedule and a parse tree of the corresponding statement in the input program. In this parse tree, each access is represent by a map mapping elements from the iteration domain to their corresponding index. Additionally, we keep track of whether the access is a read or a write (or both). The name of the iteration domain may be specified by a label on the statement. Otherwise, the name is generated to be of the form `S_`i. Each statement keeps track of its part of a global schedule. The entire global schedule corresponds to the original execution order.

As a trivial example, a dump of the representation of the program in Figure 1 extracted by `pet` is as follows.

```
context: '[N] -> { : N >= -1 }'
arrays:
- context: '[N] -> { : N >= -1 }'
  extent: '[N] -> { a[i0] : i0 >= 0 and i0 <= N }'
  element_type: int
statements:
- line: 6
  domain: '[N] -> { U[i] : i >= 0 and i <= N }'
  schedule: '[N] -> { U[i] -> [0, i] }'
  body:
    type: binary
    operation: +=
    arguments:
    - type: access
      relation: '[N] -> { U[i] -> a[i] }'
      read: 1
      write: 1
    - type: access
      relation: '[N] -> { U[i] -> [i] }'
      read: 1
      write: 0
```

The model is constructed using a bottom-up approach. A separate scop is created for each individual expression statement. The iteration domains of these statements are initially zero-dimensional, but may refer to parameters. Sequences of scops are grouped into larger scops. If such a sequence appears inside the body of the then or else branch of an `if`

statement, the iteration domains are intersected with the condition or its negation. If it appears inside the body of a loop, the parameter corresponding to the induction variable is turned into an extra dimension of the iteration domains. Additionally, we keep track of assignments to scalar variables in a simple top-down fashion. If a scalar has been assigned a known and affine expression, then this expression is substituted for the scalar inside index expressions. A scalar that has been assigned some (known or unknown) value may not be used as a parameter, but is instead considered as a data-dependent construct.

3. BASIC CONSTRUCTS

In this section we describe the basic constructs that make up a polyhedral model, i.e., the access relations that appear in expression statements, the iteration domains (constructed from `if` statements and `for` loops) and the schedules.

3.1 Access Relations

An access relation maps an iteration vector to one or more array elements and is represented by an `isl_map`. This means that the array elements are described by affine constraints, possibly involving existentially quantified variables and parameters. Since the model is constructed bottom-up, the initially constructed access relations have a zero-dimensional domain and therefore do not involve any iterators. Instead, some of the initial parameters may be converted to iterators at a later stage.

The access relation is constructed by considering each index expression individually, one for each dimension of the accessed array. Each of these index expressions is first recursively constructed as an `isl_pw_aff`, i.e., a piecewise quasi-affine expression. These objects are then converted into maps and combined into a single map. Each index expression may involve integer constants, parameters and the following operators: `+`, `-` (both unary and binary), `*`, `/`, `%` and the ternary `?:` operator. The second argument of the `/` and the `%` operators is required to be a (positive) integer literal, while at least one of the arguments of the `*` operator is required to be piecewise constant expression. For example, an index expression of the form `i * (i < 5 ? 2 : 1)` is allowed, while `(i > 10 ? i : 1) * (i < 5 ? i : 1)` is not, even though it is equivalent to `(i > 10 || i < 5) ? i : 1`. The first argument of the `?:` operator needs to satisfy the requirements of Section 3.2.

Each of the above operators has a corresponding operation in `isl` on `isl_pw_aff`s and therefore requires no extra computations in `pet`. The only exceptions are the `/` and `%` operators. `isl` does not provide any operation that directly corresponds to the C integer division (which rounds to zero), but instead provides a "floor" (which rounds to negative infinity) and a "ceil" operation (which rounds to infinity). An expression of the form `a / b` is therefore constructed as `a >= 0 ? floord(a,b) : ceild(a,b)`, with `floord` and `ceild` functions that correspond to the floor and ceil operations applied to the quotient of the arguments. The `%` is treated in a similar way. For compatibility with `CLooG` (see Section 4.4), the functions `floord` and `ceild` are also accepted inside the scop, even if they are not explicitly defined in the input. Similarly, the functions `min` and `max` are also allowed.

Depending on how a statement accesses memory `pet` marks the array accesses as read-only, write-only or a combined read and write. In function calls it is also possible to pass (the address of) an entire array or array slice as a parameter, e.g., `f(A)` or `f(A[i])` in case of a two-dimensional array `A`. In such cases, the access relation is constructed to read/write the entire array (slice).

The input program may also contain affine expressions that are not used as index expressions, but that are instead used directly as arguments to functions calls or operators. Since a program transformation may change the iterators in these expressions, we also represent them as access relation. Since there is no array involved, the range of these relations is a nameless one-dimensional space. This space can be thought of as the set of integers, with element i having value i.

3.2 Conditions

A condition is represented by an `isl_set` containing those elements that satisfy the condition. The input expression may be any boolean expression involving the `&&`, `||` and `!` operators and comparisons. A comparison is an expression that applies one of `<`, `<=`, `>`, `>=`, `==` or `!=` to two affine expressions. Such an affine expression needs to satisfy the same requirements as the index expressions in Section 3.1. An affine expression `e` itself may also be used where a comparison is expected, in which case it is treated as the comparison `e != 0`. As before, each of the above operations has a direct counterpart in `isl`.

3.3 Loops

Currently, `pet` only supports `for` loops. The only exception is the infinite loop, which may be written as either `for (;;)` or `while (1)`. The contribution of an infinite loop to the iteration domain is of the form $\{ t \mid t \geq 0 \}$. Recall that the iteration domains are constructed bottom-up and that each enclosing loop prepends a dimension to the iteration domain.

Since `pet` does not yet perform any induction variable recognition, the induction variable needs to be explicitly available in the `for` loop. That is, the loop needs to be of the form `for (i = init(n); condition(n,i); i += v)`, where `n` is any number of parameters. In particular, the initialization part needs to assign an expression to a single variable (or initialize a single newly declared variable) and this same variable needs to be incremented by a (signed) constant in the increment part. The increment may also be written `i -= -v`, `i = i + v`, `++i` or `--i` (in case `v` is 1 or -1).

The condition may be any condition that satisfies the requirements of Section 3.2. Note in particular that this means that the condition may *not* involve any variables that are being written inside the loop body as they are not considered to be parameters. In principle, the condition does not need to involve the induction variable, but such a condition will result in either an empty or an infinite loop. Let us now assume that `v` is positive. A minor variation of the construction below is used when `v` is negative. The constraints on the loop iterator imposed by the initialization and the stride can be expressed as $D = \{ i \mid \exists \alpha : \alpha \geq 0 \wedge i = \texttt{init}(\mathbf{n}) + \alpha v \}$. If `v` is equal to 1, this is simplified to $D = \{ i \mid i \geq \texttt{init}(\mathbf{n}) \}$. This simplification is performed by `pet` and does not have

any effect on the constructed iteration domain, but it may result in a simpler *representation* of this iteration domain.

As to the condition of the `for` loop, a value of the iterator belongs to the iteration domain if the condition is satisfied by that value *and* all previous values. Define the set $C = \{ i \mid \texttt{condition}(\mathbf{n}, i) \}$. The contribution of the loop to the iteration domain is the set $\{ i \in D \mid \forall i' \in D : i' \leq i \implies i' \in C \}$, or, in other words, $\{ i \in D \mid \neg\exists i' \in D : i' \leq i \wedge i' \notin C \}$. In `isl`, we can compute this set as

$$D \setminus \left(\{ i' \to i \mid i' \leq i \} (D \setminus C) \right).$$

That is, we take the elements in D that do not satisfy the condition (C), map them to later iterations and subtract those later iterations from D. If `condition(n,i)` does not involve any lower bounds on `i` then any condition satisfied by i is also satisfied by earlier iterations and the above computation can be simplified to $D \cap C$. This simplification may again result in a simpler representation of the result.

3.4 Schedule

As explained before, each statement maintains its part of the global schedule. These parts of the schedule are constructed together with the iteration domains. The initial iteration domains are zero-dimensional and the schedule simply maps this domain to a nameless zero-dimensional space. If a statement appears in a sequence of statements, the schedule for these statements is extended with an initial constant range (i.e., schedule) dimension. The values of these dimensions correspond to the order of the statements in the sequence. If a statement appears as the body of a loop, then the schedule is extended with both an initial domain dimension and an initial range dimension. If the increment on the loop is positive then these new dimensions are equated. Otherwise they are made to be opposite. That is, the schedule is extended with either $\{ i \to i \}$ or $\{ i \to -i \}$.

4. ADDITIONAL FEATURES

Besides the basic constructs described above, `pet` also supports some additional features. Some of these involve data dependent constructs and are used by the equivalence checker [33] or to construct dynamic polyhedral process networks. Others are needed to be able to parse `CLooG` output or to properly handle unsigned integers.

4.1 Data Dependent Accesses

According to the requirements of Section 3.1, an access of the form `A[i + 1 + in2[i]]` is not allowed because the index expression contains an array access. It is however still possible to perform dataflow analysis on programs containing such constructs, either by applying fuzzy dataflow analysis [5, 8] or, in the worst case, assuming that `A` may be accessed for any value of `in2`. It is even possible in some cases to prove the equivalence of two programs containing such constructs [33]. We therefore need a way of representing such data dependent accesses.

A "standard" access relation maps an element of the iteration domain to one or more array elements. In a data dependent access, the accessed array element does not only depend on the value of the parameters and the iterators, but also on the values of the nested accesses. We therefore extend the domains of the access relations to refer not only to the iterators,

```
for (i = 0; i < N; ++i)
    if (i + in2[i] >= 0 && i + in2[i] < N)
        C[i] = f(A[i + in2[i]]);
    else
        C[i] = 0;
```

Figure 2: Data dependent assignment

but also to these values. We do so by exploiting the concept of a "wrapped" relation in `isl`, which essentially allows a relation to be treated as a set while retaining information about the domain and range of the relation. In practice, we set the domain of the access relation to be a wrapped relation mapping iteration domain elements to the values of the nested accesses. For example, assume that the access above appears in statement `S_4`, then the access relation is represented as `{ [S_4[i] -> [i1]] -> A[1 + i + i1] }`. `i1` represents here the unknown result of the nested access. In addition, we keep track of the nested accesses in a list, one for each dimension in the range of the wrapped relation. In the example, this list would contain the access relation `{ S_4[i] -> in2[i] }`.

4.2 Data Dependent Assignments

Consider the program fragment in Figure 2. Not only does this program contain a data dependent access, this access is also governed by a data dependent condition. The purpose of this condition is to avoid an out-of-bounds array access and represents a fairly common idiom. In principle, this condition could be represented using the techniques of Section 4.3, but in this case there is no need to resort to these techniques. Since the conditions are only used to restrict the access relation, it seems more natural to include these conditions directly in the relation.

What is special about the idiom in Figure 2 is that the same array element is assigned in both branches of the test. We can therefore convert it to a single assignment statement using the conditional operator. That is, we convert "`if (c) a = e; else a = f;`" to "`a = c ? e : f`". Furthermore, we include the condition `c` in each access relation of `e` and its negation in each access relation of `f`. In the end, the access to `A` is represented as

```
{ [S_4[i] -> [i1]] -> A[i + i1] :
       i1 >= -i and i1 <= -1 + N - i }
```

4.3 Data Dependent Conditions

Data dependent conditions are handled in a way that is very similar to the way data dependent accesses are handled. Again, we add extra dimensions representing the values of the nested accesses to the range of a wrapped relation. In particular, they are added to the iteration domain itself. Note though that the access relations and schedules, which have the iteration domain as their domain, are not affected by this change. If the iteration domain is a wrapped relation then the domains of the schedule and the access relations refer to the domain of the wrapped relation.

The nested accesses may appear both in the condition of an `if` statement and in the condition of a `for` loop. However,

```
for (c1=ceild(n,3);c1<=floord(2*n,3);c1++) {
  for (c2=0;c2<=n-1;c2++) {
    for (j=max(1,3*c1-n);j<=min(n,3*c1-n+4);j++) {
      p = max(ceild(3*c1-j,3),ceild(n-2,3));
      if (p <= min(floord(n,3),floord(3*c1-j+2,3))) {
        S2(c2+1,j,0,p,c1-p);
      }
    }
  }
}
```

Figure 3: Part of CLooG output for thomasset test case

the condition of a `for` loop is currently not allowed to access any variables that may be written inside the loop body. The reason for this restriction is that we would have to encode that the condition is not only satisfied for the given iteration, but also for all previous iterations.

The condition of an `if` statement may also involve function calls or non-affine constructs. In this case, a separate statement is created that evaluates the condition and subsequently writes the result to a virtual array, which is marked as only attaining the values 0 and 1. The original condition is replaced by a data dependent access to this virtual array, the value of which is required to be 1 in the iteration domains of the statements in the then branch and 0 in those of the else branch. This construction is similar to the way `if` statements are handled in [14] and also to the control predication of [9].

4.4 CLooG Specific Features

The output of `CLooG` may contain special functions and constructs that require special care. Most of these have been mentioned before, but here we provide further details. First of all, as explained in Section 3.1, the output may contain the "operators" `floord`, `ceild`, `min` and `max`. Although macro definitions can be provided for these operators, it is more efficient to recognize them directly inside `pet`. For example, a macro definition for `floord` would have to encode this operation in terms of integer divisions and would therefore have to introduce several cases, while there is no need for different cases if it is recognized directly. Similarly, if the condition of a for loop is of the form `i <= min(a,b)`, it can be directly encoded as $\{ i \mid i \leq a \wedge i \leq b \}$ instead of introducing cases depending on the difference between `a` and `b`. This is especially important if there are many nested `min`s or `max`s as in the `classen2` test case. Finally, the simple forward substitution of scalars discussed in Section 2 is also essential for parsing some `CLooG` outputs. Consider for example part of the output for the `thomasset` test case, reproduced in Figure 3. At first sight, the `if` statement looks like it involves a data-dependent condition, but by plugging in the expression assigned to `p` in the previous statement, it can be analyzed as a static affine condition.

4.5 Support for Unsigned Integers

In C99 signed and unsigned integer types do not only define different sets of values, but they also behave differently. Signed values yield undefined behavior if the result of an expression is not within the range of representable values (C99 6.5). Like other compilers pet can and does assume that undefined behavior is never triggered by a valid program. Consequently it assumes that for signed types the results of all expressions fit in the corresponding type. This means pet can directly translate such expressions to `isl_pw_aff` expressions.

```
for (unsigned i = 0; i < n; i++)
        A[0] += i;

for (unsigned j = 0; j < n + 1; j++)
        B[0] += j;
```

Figure 4: Unsigned operation in loop bound

```
for (unsigned i = 0; i < n; i++) {
        A[0] += i;
        B[0] += i;
}

B[0] += n+1;
```

Figure 5: Invalid fusion of program in Figure 4

For unsigned types C99 6.2.5 includes the following exception: *"[..] a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type"*. The program in Figure 4 consists of two loops with n and $n+1$ iterations. If the loop bounds are represented as $i \leq n$ and $j \leq n+1$, fusing the two loops to the code in Figure 5 is possible. Yet, this is invalid in the presence of integer wrapping. If n is the maximal unsigned value, the expression $n+1$ will evaluate to 0 such that in the original code B[0] is not accessed at all. However, the transformed code accesses B[0] many times.

To ensure correctness it is necessary to perform all unsigned operations modulo the number of elements in the integer type. This means for the example above that the loop bounds should not be n and $n+1$, but n mod (UINT_MAX+1)) and $(n+1)$ mod (UINT_MAX + 1). `pet` knows about the types of variables and automatically introduces the necessary modulo operations.

Let us now consider in a bit more detail how an unsigned loop iterator affects the way the iteration domain and the schedule are constructed. The iteration domain is first constructed as explained in Section 3.3, but in terms of a virtual loop iterator, with the condition of the loop changed to apply to the modulo of this virtual loop iterator instead of the virtual iterator itself. Afterwards, a mapping is applied to the iteration domain and the domain of the schedule that

```
for (unsigned char k=252; (k%9) <= 5; ++k)
        S:;
```

Figure 6: Loop with unsigned iterator

wraps the virtual iterator to the real iterator, but only after intersecting the domain of the schedule with the iteration domain. This intersection is needed to ensure that we do not lose any information as some iterations in the wrapped domain may be scheduled several times, typically an infinite number of times. As an example of a loop with an unsigned iterator, consider the loop in Figure 6. The corresponding domain and schedule are as follows.

```
domain: '{ S[k] : exists
 (e0 = [(507 - k)/256]: k >= 0 and
  k <= 255 and 256e0 >= 252 - k and
  256e0 <= 261 - k) }'
schedule: '{ S[k] -> [0, o1] : exists
 (e0 = [(-k + o1)/256]: 256e0 = -k + o1 and
  o1 >= 252 and k <= 255 and k >= 0 and
  o1 <= 261)
```

It may be difficult to see, but this loop has 10 iterations, first from 252 to 255 and then from 0 to 5. Applying the `scan` operator to the schedule in `iscc` makes this clear:

```
{ S[5] -> [0, 261]; S[4] -> [0, 260];
  S[3] -> [0, 259]; S[2] -> [0, 258];
  S[1] -> [0, 257]; S[0] -> [0, 256];
  S[255] -> [0, 255]; S[254] -> [0, 254];
  S[253] -> [0, 253]; S[252] -> [0, 252] }
```

5. RELATED WORK

We are aware of several proprietary compilers, including ATOMIUM [12], R-Stream [26], Cosy [22] and IBM-XL [10], that use polyhedral techniques. As these systems are not available to us and there is little documentation on their polyhedral model extractors, we cannot perform any detailed comparison with these compilers. At least two polyhedral optimizers, Bee [1] and PolyOpt [25], have been developed for the ROSE compiler. Unfortunately, these systems do not appear to be publicly available. It is known however that unlike clang, ROSE does not fully support C99. In particular, ROSE does not support VLAs. Moreover, judging from the documentation [25], PolyOpt imposes somewhat severe restrictions on the allowed iteration domains, in particular requiring them to be convex. This means for example that `else` and `!=` are not supported. On the other hand, the system does include an optimization engine and code synthesis. The insieme compiler[3] is reported to use an extraction tool that is somewhat similar to `pet` in that it also uses clang for parsing C code. It does however *not* use `isl` to construct and represent the polyhedral model, but instead a custom constraints based representation. The supported features appear to be similar to those supported by `clan`.

The Omega Project contains a dependence analysis tool called petit [19]. Although it does not appear to be possible to have petit dump a polyhedral model, the tool necessarily does include a parser. The input language is similar to Fortran and the parser includes some advanced features such as induction variable recognition and forward substitution of scalars. Like the `pet` predecessor `pers`, CHiLL [13] uses SUIF for parsing, whence no support for C99, and appears to handle even fewer constructs than `pers`. The LooPo [16] project includes a polyhedral parser, which accepts subsets of both C and Fortran (or a combination) as input. Index expressions are required to be affine (rather than piecewise quasi-affine), but there is support for generic `while` loops [14]. The parser that comes with FADAlib [8] also supports `while` loops, but does not support many of the constructs supported by `pet`. PIPS [3] also performs polyhedral analysis, but mostly in the sense of abstract interpretation. Although earlier versions allowed for the extraction of a polyhedral model in our sense, i.e., with access relations and iteration domains, this functionality appears no longer to be supported.

Several open source compilers have support for extracting a polyhedral model from an internal representation, including WRaP-IT [15] in ORC, graphite [24] in GCC and Polly [17] in LLVM/clang. As explained in the introduction, such low-level parsers have their advantages and disadvantages when compared to source level parsers such as `pet`. In Table 1, we perform a more detailed comparison with Polly, which is based on the same compiler infrastructure as `pet`. The table also compares against `clan` [7], which to the best of our knowledge is currently the most popular source level parser. There are however many different versions of `clan`, including some such as irClan [9] that include some support for data dependent constructs. This irClan system does not appear to be publicly available though. Here, we compare against the latest official release (version 0.6.0).

6. LIMITATIONS AND FUTURE WORK

Although `pet` is already very powerful, it is still fairly new and therefore suffers from some limitations. Perhaps the most prominent limitation is that it currently only extracts a polyhedral model and does not perform any code synthesis. As suggested by its name, this was also originally the intent. Clearly, the equivalence checker does not need to perform any code synthesis. Also, when constructing process networks, each statement is assumed to call a single function and this same function is then also called from the corresponding process, which is generated on hardware. For PPCG, code synthesis would be useful, but the use of `pet` is already an improvement over `clan` as `pet` provides a parse tree of each statement, whereas `clan` only provides a string, which has to be parsed again if some of the accesses need to be changed. Since `clang` has good support for "pretty printing" we expect that performing code synthesis again after transformation would be relatively easy.

The set of acceptable input could be expanded by performing more extensive analysis to obtain more relations between variables. At the moment, we only perform a very simple form of forward substitution. We would like to add induction variable recognition, some limited form of abstract interpretation to detect affine relation and perhaps some (fuzzy) array dataflow analysis.

There is currently no support for `switch` statements, although it should be fairly easy to add. Support for `break` and `continue` statements may also be useful and ties in with allowing more generic loop conditions, including those that involve variables written inside the body of the loop. A proper treatment may require support for uninterpreted functions in `isl`. We are currently experimenting with them

[3] `http://www.dps.uibk.ac.at/insieme/index.html`

Feature	Polly	clan	pet
General			
Scop Detection (auto)	yes	no	yes
Scop Detection (pragma)	no	yes	yes
Highlight unsupported code	yes[a]	no	yes
Parse entire source files	yes	yes[b]	yes
Parse isolated scops	no	yes	no
Input language	various[c]	C-like	C99
Code synthesis	yes	no	no
Classical Scop			
Affine Expressions			
- add, multiply	yes	yes	yes
- max	yes	yes	yes
- min	no	yes	yes
- modulo	no	no	yes
- division	yes	no	yes
- floord/ceild	no	yes	yes
- conditional (a ? b : c)	no	no	yes
Comparisons			
- $<, \leq, =, \geq, >$	yes	yes	yes
- $\neq$	yes	no	yes
- Implicit Comparison to Zero	yes	no	yes
Boolean Combinations			
- and (&&)	no	yes	yes
- or (||)	no	no	yes
- not (!)	no	no	yes
Loops			
- for-loop	yes	yes	yes
- Stride > 1	yes	no	yes
- Negative Stride	yes	no	yes
- Loop bound restriction	SCEV[d]	syntactic	isl
Conditions			
- if	yes	yes	yes
- else	yes	no	yes
Memory Access			
- Array (Static Size)	yes	yes	yes
- Array (Variable Size)	no	yes	yes
- Pointer Arithmetic	yes	no	no
Parse any CLooG Output	no	no[e]	yes
Semantic Analysis			
Propagate Expressions	yes	no	yes
Recognize Induction Variables	yes	no	no
Semantic Loops	yes	no	no
Alias Analysis	yes	no	no
Extensions			
Derive Array Sizes	no	no	yes
Data Dependent Access	no	no	yes
Data Dependent Assignment	no	no	yes
Data Dependent Condition	no	no	yes
Infinite Loops	no	no	yes
Wrapping (Unsigned Ops)	no	no	yes
Wrapping (Casts)	no	no	no
Inlining	yes	no	no

[a] On intermediate language
[b] clan ignores everything outside the scop
[c] Any language that can be lowered to LLVM IR.
[d] Scalar Evolution Analysis built around the ideas in [28]
[e] For example, the fragment in Figure 3

Table 1: Features of different polyhedral extractors

outside of `isl` to simplify the results of fuzzy dataflow analysis.

Other issues include the following. Like most high-level polyhedral model extractors, `pet` does not perform any alias analysis, but instead assumes that none of the arrays accessed in the scop overlap. `pet` is based on `clang`, such that conceptually C, C++ or Objective-C code can be analyzed. At the moment only C is supported. Other languages such as Fortran are not supported and are unlikely to ever be supported. While `pet` has fairly good support for *unsigned* integers, support for *signed* integers could be improved. In particular, we should detect undefined behavior and update the context accordingly. Another area where `pet` can be improved is that of (implicit) casts. Support for them is not yet available, even though adding this support should not impose any difficulties. The AST generated by `clang` contains the relevant information and the use of `isl` makes adding the relevant modulo operations trivial.

7. CONCLUSIONS

By exploiting the strengths of both `clang` and `isl`, we have constructed a new polyhedral extraction tool with several advanced features. To the best of our knowledge, `pet` has the most extensive support for static piecewise quasi-affine index expressions and conditions. The resulting access relations, iteration domains and schedules can be used directly by any sufficiently generic dataflow analysis implementation, e.g., that in `isl`. Although `pet` is still missing some features such as alias analysis, which may in some cases lead to incorrect results, our support for unsigned integers shows that we are committed to correctness. At the same time, our support for CLooG specific features shows a sense of pragmatism. Finally, we support some data dependent constructs that are being used in practice by an equivalence checker and/or a tool for deriving dynamic polyhedral process networks.

8. ACKNOWLEDGMENTS

This work was partially funded by a gift received by LIACS from Intel Corporation.

9. REFERENCES

[1] C. Alias, F. Baray, and A. Darte. Bee+cl@k: an implementation of lattice-based array contraction in the source-to-source translator rose. *SIGPLAN Not.*, 42(7):73–82, 2007.

[2] S. Amarasinghe, J. Anderson, M. S. Lam, and C.-W. Tseng. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995.

[3] M. Amini, C. Ancourt, F. Coelho, F. Irigoin, P. Jouvelot, R. Keryell, P. Villalon, B. Creusillet, and S. Guelton. PIPS is not (just) polyhedral software. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[4] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[5] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput.*, 40(2):210–226, 1997.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[7] C. Bastoul. Clan - a polyhedral representation extractor for high level programs, May 2008.

[8] M. Belaoucha, D. Barthou, A. Eliche, and S. A. A. Touati. FADAlib: an open source C++ library for fuzzy array dataflow analysis. *Procedia CS*, 1(1):2075–2084, 2010.

[9] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, Paphos, Cyprus, Mar. 2010. Springer-Verlag.

[10] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008.

[12] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergale, and A. Vandecapelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia Design.* Kluwer Academic Publishers, 1998.

[13] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, USC Department of Computer Science, June 2008.

[14] M. Geigl. Parallelization of loop nests with general bounds in the polyhedron model. Master's thesis, Mar. 1997.

[15] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[16] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In *Proc. Sixth Workshop on Compilers for Parallel Computers, volume 21 of Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum, 1996.

[17] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - polyhedral optimization in llvm. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[18] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[19] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. New user interface for petit and other interfaces: user guide. Technical report, University of Maryland, Dec. 1996.

[20] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library. Technical report, University of Maryland, Nov. 1996.

[21] V. Loechner. PolyLib: A library for manipulating parameterized polyhedra. Technical report, Mar. 1999.

[22] S. Meijer and M. Beemster. Parallelization using polyhedral analysis, Mar. 2008. White Paper, ACE Associated Compiler Experts BV, Amsterdam.

[23] D. Nadezhkin, H. Nikolov, and T. Stefanov. Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In *ESTImedia*, pages 21–30. IEEE, 2010.

[24] S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. Technical Report A/378/CRI, Centre de Recherche en Informatique, École des Mines de Paris, Fontainebleau, France, 2006. Contribution to the GNU Compilers Collection Developers Summit 2006 (GCC Summit 06), Ottawa, Canada, June 28–30, 2006.

[25] L.-N. Pouchet. Polyopt, a polyhedral optimizer for the rose compiler, July 2011.

[26] E. Schweitz, R. Lethin, A. Leung, and B. Meister. R-stream: A parametric high level compiler. In J. Kepner, editor, *Proceedings of HPEC 2006, 10th annual workshop on High Performance Embedded Computing*, Lincoln Labs, Lexington, MA, 2006.

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

[28] R. A. Van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *In Proceedings of the International Conference on Compiler Construction*, ETAPS CC 'O1, pages 118–132, 2001.

[29] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

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

[31] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pages 599–613. Springer, June 2009.

[32] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, special issue on Embedded Digital Signal Processing Systems*, 2007, 2007.

[33] S. Verdoolaege, M. Palkovic, M. Bruynooghe, G. Janssens, and F. Catthoor. Experience with widening based equivalence checking in realistic multimedia systems. *Journal of Electronic Testing*, 26(2):279–292, 2010.