PolyOpt

A Polyhedral Optimizer for the ROSE compiler
Edition 0.1, for PolyOpt 0.1.0
July 1st 2011

Louis-Noël Pouchet

This manual is dedicated to PolyOpt version 0.1.0, a framework for Polyhedral Optimization in the ROSE compiler.

Table of Contents

1 Introduction

PolyOpt is a polyhedral loop optimization framework, integrated in the ROSE compiler. The main features are:

- Automatic extraction of regions that can be optimized in the polyhedral model
- Full support of PoCC (the Polyhedral Compiler Collection) analysis and optimizations
  - Dependence analysis with Candl
  - Program transformations for tiling and parallelism with Pluto
  - Code generation with CLooG
  - Parametric tiling with PTile
  - Numerous under-the-hood functionalities and optimizations

Note, only a subset of C is currently supported by PolyOpt: C++ and Fortran programs are not supported.

Communication: Please contact directly Louis-Noel Pouchet pouchet@cse.ohio-state.edu for any question. PoCC is also available as a stand-alone software on sourceforge

2 Specifics of Polyhedral Programs

2.1 Static Control Parts

Sequences of (possibly imperfectly nested) loops amenable to polyhedral optimization are called *static control parts* (SCoP) [5], roughly defined as a set of consecutive statements such that all loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters (variables whose value is unknown at compile time but invariant in the loop nest considered). In addition, for effective data dependence analysis we require the array access functions to also be affine functions of the surrounding iterators and parameters.

For instance, a valid affine expression for a conditional or a loop bound in a SCoP with three loops iterators *i,j,k* and two parameters *N,P* will be of the form *a.i* + *b.j* + *c.k* + *d.N* + *e.P* + *f*, *a,b,c,d,e,f* are arbitrary (possibly 0) integer numbers.

The following program is a SCoP:

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        A[i][j] = A[i][j] + u1[i]*v1[j]
        if (N - i > 2)
           A[i][j] -= 2;
     }
```

Numerous elements can break the SCoP property, for instance:

- `if` conditionals involving variables that are not a loop iterator or a parameter, e.g., `if (A[i][j] == 0)`.
- `if` conditionals involving loop iterators and/or a parameter to form a non-affine expression, e.g., `if (i * j == 0)`.
- Non-affine `for` initialization or test condition, e.g., `for (j = 0; j < i * i; ++i)`.
- Non-affine array access, e.g., `A[i*N][j % i]` or `A[B[i]]`.

2.2 Additional Restrictions in PolyOpt

PolyOpt automatically extracts maximal regions that can be optimized in the Polyhedral framework. We enforce numerous additional constraints to ensure the correctness of the SCoP extraction, in particular due to dependence analysis consideration:

- The only allowed control statements are `for` and `if`.
- There is no function call in the SCoP.

- There is no explicit pointer arithmetic/manipulation in the SCoP (no `&` nor `*` operators).
- `goto`, `break` and `continue` statements are forbidden.
- Arrays represent distinct memory locations (one per accessed array cell), and arrays are not aliased (note: **no check is performed by PolyOpt for this property, it is the responsibility of the user to not feed ill-formed arrays**).
- Loops increment by step of one.

2.3 Allowed Control-Flow Operations

PolyOpt supports a wide range of affine expressions, in particular conjunctions of affine constraints can be used to bound the space. In all the following, we recall that an affine expression must involve only surrounding loop iterators and parameters (scalar variables that are invariant during the SCoP execution).

SCoP extraction is a syntactic process so there are clear definitions of what is allowed in `for (init; test; increment)` and `if (condition)` statements. We note that if the loop iterator of a `for` statement is used outside the scope of the loop, or is assigned in the loop body, the *loop will conservatively not be considered for SCoP extraction* since PolyOpt may change the exit value of loop iterators.

2.3.1 In `for` initialization statement

`init` can be either empty, or of the from `<type> var = expressionLb`. That is, a single variable initialization is supported. The `expressionLb` is an affine expression, or possibly a conjunction of expressions with the `max(expr1, expr2)` operator. The `max` operator can either be written using a call to the `max` function together with using the `--polyopt-safe-math-func` flag, or with the ternary operation `a < b ? b : a`.

If the loop has no lower bound, the polyhedral representation will assume an infinite lower bound for the loop iterator: no analysis is performed to determine if there exists an initialization of the loop iterator before the `for` statement.

As an illustration, all loops in the following code form a valid SCoP.

```
for (int i = max(max(N,M),P); i < N; i++)
    for (j = max(i + 2 + 3*M, Q); j < N; j++)
       for (k = i - 2 < 0 ? i - 2 : 0); k < N; k++)
           for (; ; l++)
               A[i][j] -= 2;
```

Some examples of incorrect loop lower bound include:

- `for (i = 0, j = 0; ...)`: more than one initialization.
- `for (i = max(a,b) + max(c,d); ...)`: not a valid conjunction.

- `for (i = max(a,b) + P; ...)`: not a valid conjunction.
- `for (i = a > b ? a : b; ...)`: not a (syntactically) valid ternary `max` operator.

2.3.2 In `for` test statement

`test` can be either empty (infinite loop), or of the from `var opComp expressionUb <&& var opComp expressionUb2 <&& ...> >`. That is, conjunction of upper bounds are supported via the `&&` operator. The `expressionUb` is an affine expression, or possibly a conjunction of expressions with the `min(expr1, expr2)` operator. The `min` operator can either be written using a call to the `min` function together with using the `--polyopt-safe-math-func` flag, or with the ternary operation `a < b ? a : b`. The `opComp` must be one of `<, <=, ==`.

As an illustration, all loops in the following code form a valid SCoP.

```
for (int i = 0; i < N && i < min(min(P,Q),R); i++)
    for (j = 0; j <= (i < P ? P : i); j++)
       for (k = 0; k <= 0; k++)
           for (; ; l++)
               A[i][j] -= 2;
```

Some examples of incorrect loop lower bound include:

- `for (i = 0; i < 1 || i < 2)`: disjunctions are not allowed.
- `for (i = 0; i < min(a,b) + min(c,d); ...)`: not a valid conjunction.
- `for (i = 0; min(i, N); ...)`: missing the `var opComp` part.
- `for (i = 0; i > P; ...)`: incorrect comparison operator.
- `for (i = 0; i < (a > b ? b : a); ...)`: not a (syntactically) valid ternary `max` operator.

2.3.3 In `for` increment statement

Loops must increment by step of one, and there must be a single operation in the `increment` part. Typically only `i++`, `++i` and `i+=1` are supported increments. More complex increments such as `i += one` or `i += 1, j += 1` are not supported.

2.3.4 In `if` conditional statement

For `if` statements, the conditional expression can be an arbitrary affine expression, and a conjunction of expressions with the `&&` operator. `min` and `max` are allowed, provided a `max` constrains the lower bound of a variable and a `min` constraints the upper bound of a variable. Most standard comparison operators are allowed: `<, <=, ==, >=, >`. Note that `else` clause is not allowed, nor is `!=`.

As an illustration, all loops in the following code form a valid SCoP.

```
   if (i > max(M,N) && j == 0)
      if (k < 32 && k < min(max(i,j),P))
         A[i][j] = 42;
```

Some examples of incorrect conditional expressions include:

- `if (i == 0 || c == 0)`: disjunctions are not allowed.
- `if (i < max(A,B))`: not a valid `max` constraint.
- `if (i == 42/5)`: not an integer term.

2.3.5 Examples

We conclude by showing some examples of SCoPs automatically detected by PolyOpt. Note that the only constraints for the statements (e.g., `R,S,T` in the next example) involve the lack of function calls, *at most one variable is assigned in the statement*, and using affine functions to dereference arrays.

```
  alpha = 43532;
  beta = 12313;
  for (int i = 0; i < N; i++) {
R:      v1[i] = (i+1)/N/4.0;
S:      w[i] = 0.0;
        for (j = 0; j < N; j++)
T:         A[i][j] = ((DATA_TYPE) i*j) / N;
    }
```

```
  for (j = 1; j <= m; j++) {
      stddev[j] = 0.0;
      for (i = 1; i <= n; i++)
       stddev[j] += (data[i][j] - mean[j]) * (data[i][j] - mean[j]);
      stddev[j] /= float_n;
      stddev[j] = sqrt(stddev[j]);
      stddev[j] = stddev[j] <= eps ? 1.0 : stddev[j];
   }
```

3 Optimization Paths

Three main optimization paths are available in PolyOpt. They are geared towards improving data locality for fewer data cache misses, and both coarse- and fine-grain shared memory parallelization with OpenMP. They will be applied on all Static Control Parts that were automatically detected in the input program. Program transformations are generated via the PoCC polyhedral engine.

3.1 `--polyopt-fixed-tiling`

3.1.1 Description

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel blocked (if possible) execution of the SCoP. The default tile size is 32, and can be specified at compile time only. Parallel loops are marked with OpenMP pragmas, inner-most vectorizable loops are marked with ivdep pragmas. Parallel or pipeline-parallel tile execution is achieved when tiling is possible.

The Pluto module is used to compute the loop transformation sequence, in the form of a series of affine multidimensional schedules.

Giving the flag `--polyopt-fixed-tiling` to PolyOpt is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-tile`
- `--polyopt-pluto-parallel`
- `--polyopt-pluto-prevector`
- `--polyopt-generate-pragmas`

3.1.2 Example

Given the input program:

```
   for (i = 0; i < n; i++)
     for (j = 0; j < n; j++) {
         C[i][j] *= beta;
         for (k = 0; k < n; k++)
           C[i][j] +=A[i][k] * B[k][j] * alpha;
}
```

One can *optionally* specify a file to set the tile sizes, to override the default *32* value. This file must be called `tile.sizes`, and be stored in the current working directory. It must contain one tile size per dimension to be tiled. For example:

```
$> cat tile.sizes
16 64 1
```

The result of `--polyopt-fixed-tiling` on the above example, with the specified `tile.sizes` file is shown below. Note, if a `tile.sizes` file exists in the current working directory it will *always* be used.

```
{
    int c6;
    int c3;
    int c1;
    int c2;
    int c4;
    if (n >= 1) {
#pragma omp parallel for private(c4, c2, c6)
      for (c1 = 0; c1 <= (((n + -1) * 16 < 0?
                  ((16 < 0?-((-(n + -1) + 16 + 1) / 16) :
                  -((-(n + -1) + 16 - 1) / 16))) : (n + -1) / 16)); c1++)
        for (c2 = 0; c2 <= (((n + -1) * 64 < 0?
                    ((64 < 0?-((-(n + -1) + 64 + 1) / 64) :
                    -((-(n + -1) + 64 - 1) / 64))) : (n + -1) / 64)); c2++)
          for (c4 = 16 * c1; c4 <= ((16 * c1 + 15 < n + -1?
                               16 * c1 + 15 : n + -1)); c4++)
#pragma ivdep
#pragma vector always
            for (c6 = 64 * c2; c6 <= ((64 * c2 + 63 < n + -1?
                                  64 * c2 + 63 : n + -1)); c6++)
              (C[c4])[c6] *= beta;
#pragma omp parallel for private(c4, c2, c3, c6)
      for (c1 = 0; c1 <= (((n + -1) * 16 < 0?
                  ((16 < 0?-((-(n + -1) + 16 + 1) / 16) :
                  -((-(n + -1) + 16 - 1) / 16))) : (n + -1) / 16)); c1++)
        for (c2 = 0; c2 <= (((n + -1) * 64 < 0?
                    ((64 < 0?-((-(n + -1) + 64 + 1) / 64) :
                    -((-(n + -1) + 64 - 1) / 64))) : (n + -1) / 64)); c2++)
          for (c3 = 0; c3 <= n + -1; c3++)
            for (c4 = 16 * c1; c4 <= ((16 * c1 + 15 < n + -1?
                                 16 * c1 + 15 : n + -1)); c4++)
#pragma ivdep
#pragma vector always
              for (c6 = 64 * c2; c6 <= ((64 * c2 + 63 < n + -1?
                                    64 * c2 + 63 : n + -1)); c6++)
                (C[c4])[c6] += ((((A[c4])[c3]) * ((B[c3])[c6])) * alpha);
    }
}
```

3.2 `--polyopt-parametric-tiling`

3.2.1 Description

NOTE: The parametric tiling path is still experimental, and correctness of the generated code is not guaranteed in all cases. In particular, a known issue is when parametric tiling is applied on a loop nest where the outer loop is sequential (wavefront creation is required) and the inner loops are permutable but not fusable. We are working hard to fix this remaining problem.

To the best of our knowledge, the generated code is correct when all statements in a (possibly imperfectly nested) loop nest can be maximally fused. Remember that polyhedral transformations are automatically computed before the parametric tiling pass to enforce this property on the code when possible. The above issue impacts only program parts where it is not possible to exhibit a polyhedral transformation making either the outer loop parallel, or all loops fusable in the loop nest. This is not a frequent pattern, for instance none of the 28 benchmarks of the PolyBench 2.0 test suite exhibit this issue.

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel blocked execution of the SCoP. The generated code is parametrically tiled when possible, and the tile sizes can be specified at runtime via the `__pace_tile_sizes[]` array. By default, the tile sizes are set to 32. Parallel loops are marked with OpenMP pragmas.

The Pluto module is used to compute a loop transformation sequence that makes tiling legal, when possible, and the PTile module performs parametric tiling. Parallel or pipeline-parallel tile execution is achieved if tiling is possible.

Giving the flag `--polyopt-parametric-tiling` to PolyOpt is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-parallel`
- `--polyopt-codegen-use-ptile`
- `--polyopt-codegen-insert-ptile-api`

3.2.2 Example

The PACE tiling API requires to use the function `PACETileSizeVectorInit(int*, int, int)` to fill-in the tile sizes. This function takes an array of integers, the number of tile size parameters, and a unique identifier for the SCoP. This function can be in another compilation unit, inserted automatically by the PACE compiler, or added manually by the user. It allows to select the tile size at run-time, before the computation starts.

The result of `--polyopt-parametric-tiling` on the above `dgemm` example is shown below.

```
{
    int ___pace_tile_sizes[3];
    PACETileSizeVectorInit(___pace_tile_sizes,3,2);
    int c2;
    int c2t1;
    int c1;
    int c3;
    int c1t1;
    float T1c3 = (float )(___pace_tile_sizes[0]);
    int c3t1;
    float T1c2 = (float )(___pace_tile_sizes[1]);
    float T1c1 = (float )(___pace_tile_sizes[2]);
    if (n >= 1) {
      {
        int tmpLb;
        int tmpUb;
        tmpLb = round(-1 + 1 / T1c1);
        tmpUb = round(n * (1 / T1c1) + 1 / T1c1 * -1);
#pragma omp parallel for private(c2t1, c1, c2)
        for (c1t1 = tmpLb; c1t1 <= tmpUb; ++c1t1)
          for (c2t1 = round(-1 + 1 / T1c2);
               c2t1 <= round(n * (1 / T1c2) + 1 / T1c2 * -1); ++c2t1)
            for (c1 = (c1t1 * T1c1 > 0?c1t1 * T1c1 : 0);
                 c1 <= ((c1t1 * T1c1 + (T1c1 + -1) < n + -1?
                 c1t1 * T1c1 + (T1c1 + -1) : n + -1)); c1++)
              for (c2 = (c2t1 * T1c2 > 0?c2t1 * T1c2 : 0);
                   c2 <= ((c2t1 * T1c2 + (T1c2 + -1) < n + -1?
                   c2t1 * T1c2 + (T1c2 + -1) : n + -1)); c2++)
                (C[c1])[c2] *= beta;
      }
      {
        int tmpLb;
        int tmpUb;
        tmpLb = round(-1 + 1 / T1c1);
        tmpUb = round(n * (1 / T1c1) + 1 / T1c1 * -1);
#pragma omp parallel for private(c2t1, c3t1, c1, c2, c3)
        for (c1t1 = tmpLb; c1t1 <= tmpUb; ++c1t1)
          for (c2t1 = round(-1 + 1 / T1c2);
               c2t1 <= round(n * (1 / T1c2) + 1 / T1c2 * -1); ++c2t1)
            for (c3t1 = round(-1 + 1 / T1c3);
                 c3t1 <= round(n * (1 / T1c3) + 1 / T1c3 * -1); ++c3t1)
              for (c1 = (c1t1 * T1c1 > 0?c1t1 * T1c1 : 0);
                   c1 <= ((c1t1 * T1c1 + (T1c1 + -1) < n + -1?
                   c1t1 * T1c1 + (T1c1 + -1) : n + -1)); c1++)
                for (c2 = (c2t1 * T1c2 > 0?c2t1 * T1c2 : 0);
                     c2 <= ((c2t1 * T1c2 + (T1c2 + -1) < n + -1?
                     c2t1 * T1c2 + (T1c2 + -1) : n + -1)); c2++)
                  for (c3 = (c3t1 * T1c3 > 0?c3t1 * T1c3 : 0);
                       c3 <= ((c3t1 * T1c3 + (T1c3 + -1) < n + -1?
                       c3t1 * T1c3 + (T1c3 + -1) : n + -1)); c3++)
                    (C[c1])[c2] += ((((A[c1])[c3]) * ((B[c3])[c2])) *alpha);
      }
   }
}
```

3.3 `--polyopt-parallel-only`

3.3.1 Description

This path automatically computes and applies a complex, SCoP-specific sequence of loop transformations to enable parallel execution of the SCoP while improving data locality. In contrast to the other paths, no tiling is applied on the generated program. Parallel loops are marked with OpenMP pragmas. The Pluto module is used to compute a loop transformation sequence that expose coarse-grain parallelism when possible.

Giving the flag `--polyopt-parallel-only` to PolyOpt is equivalent to giving the sequence:

- `--polyopt-pluto-fuse-smartfuse`
- `--polyopt-pluto-parallel`
- `--polyopt-generate-pragmas`

3.3.2 Example

The result of `--polyopt-parallel-only` on the above `dgemm` example is shown below. Note that pre-vectorization is disabled in this mode, fixed tiling must be enabled for it to be active. To prevent the distribution of the two statements, the user can rely on the fine-tuning flags, e.g., `--polyopt-pluto-fuse-maxfuse`.

```
{
    int c2;
    int c1;
    int c3;
    if (n >= 1) {
#pragma omp parallel for private(c2)
      for (c1 = 0; c1 <= n + -1; c1++)
        for (c2 = 0; c2 <= n + -1; c2++)
          (C[c1])[c2] *= beta;
#pragma omp parallel for private(c3, c2)
      for (c1 = 0; c1 <= n + -1; c1++)
        for (c2 = 0; c2 <= n + -1; c2++)
          for (c3 = 0; c3 <= n + -1; c3++)
            (C[c1])[c2] += ((((A[c1])[c3]) * ((B[c3])[c2])) * alpha);
    }
}
```

4 Fine-tuning Optimizations

PolyOpt offer numerous tuning possibilities, use `--polyopt-help` for a comprehensive list. We distinguish two main categories of options that impact how the program will be transformed: (1) options that control how SCoP are extracted; and (2) options that control how each individual SCoP is transformed.

4.1 SCoP Detection

The following options are available to control how SCoP extraction is being performed, and in particular how non-compliant features are handled.

- `--polyopt-safe-math-func`: Consider function calls whose prototype is declared in math.h (e.g., round, sqrt, etc.) as side-effect free functions, meaning a call to one of these functions will not break the SCoP.
- `--polyopt-approximate-scop-extractor`: Over-approximate non-affine array accesses to scalars (all array cells are approximated to be read/written for each array reference).
- `--polyopt-scop-extractor-verbose=1`: Verbosity option. Reports which functions have been analyzed.
- `--polyopt-scop-extractor-verbose=2`: Verbosity option. Reports which SCoPs have been detected.
- `--polyopt-scop-extractor-verbose=3`: Verbosity option. Reports which SCoPs have been detected.
- `--polyopt-scop-extractor-verbose=4`: Verbosity option. Reports which SCoPs have been detected, print their polyhedral representation, print all nodes that broke the SCoP.

4.2 Tuning Optimizations

The following options are available to control PoCC, the polyhedral engine. In particular, those control the Pluto module that is responsible for computing the loop transformation to be applied to the SCoP.

- `--polyopt-pocc-verbose`:
- `--polyopt-pluto`: Activate the Pluto module.
- `--polyopt-pluto-tile`: Activate polyhedral tiling.
- `--polyopt-pluto-parallel`: Activate coarse-grain parallelization.
- `--polyopt-pluto-prevector`: Activate fine-grain parallelization.
- `--polyopt-pluto-fuse-<maxfuse,smartfuse,nofuse>`: Control which fusion heuristic to use (default is smartfuse).
- `--polyopt-pluto-rar`: Consider Read-After-Read dependences for improved data locality.

- `--polyopt-pluto-ft <value>`: Control the first level at which CLooG will start to optimize.
- `--polyopt-pluto-lt <value>`: Control the last level at which CLooG will start to optimize.

5 Troubleshooting

In PolyOpt, polyhedral programs are a constrained subset of C programs and it can be difficult at start to understand why a program is not detected as a SCoP. Try using the `--polyopt-scop-extractor-verbose=4` option, and reading the papers referenced below.

For any other problem, please contact directly Louis-Noel Pouchet pouchet@cse.ohio-state.edu.

6 References

[**1**] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *International Symposium on Code Generation and Optimization (CGO'10)*, Apr 2010.

[**2**] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, Sept 2004.

[**3**] Uday Bondhugula and Albert Hartono and J. Ramanujam and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, Jun 2008.

[**4**] Paul Feautrier. Dataflow Analysis of Array and Scalar References. In *Intl. Journal of Parallel Programming*, 20(1):23–53, 1991.

[**5**] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II, Multidimensional time. In *Intl. Journal of Parallel Programming*, 21(5):389–420, 1992.

[**6**] Louis-Noel Pouchet, Uday Bondhugula, Cdric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*, Jan 2011.