SCoP Detection: A Fast Algorithm for Industrial Compilers

Aditya Kumar
Samsung Austin R&D Center
aditya.k7@samsung.com

Sebastian Pop
Samsung Austin R&D Center
s.pop@samsung.com

ABSTRACT

SCoP detection is a search algorithm that a compiler for imperative programming languages is using to find loops to be represented and optimized in the polyhedral model.

We improved the current algorithms for SCoP detection by operating on the natural loops tree, a higher level representation of the CFG, in order to lower the overall compilation time. The algorithm described in this paper has been implemented in GCC 6.0 as a requirement to enable by default the isl schedule optimizer at "-O3 -fprofile-use".

We present evidence that the new SCoP detection algorithm improves the overall compilation time: on a large C++ application, the overall compilation time spent in SCoP detection was reduced from 7% to 0.3%. Experimental results also show that GCC detects larger SCoPs on Polybench: 6.09 loops per SCoP as compared to 2.59 loops per SCoP with the previous algorithm.

1. INTRODUCTION

Loop optimizations are usually described for languages like Fortran where loops and arrays are well-behaved syntactic constructs. In contrast with Fortran, low-level languages like C or C++ do not offer similar ease in the analysis of loops, induction variables, memory references, and data dependences, which are essential to all high-level loop transforms. Compilers for low-level languages like GCC and LLVM lower the statements and expressions into low-level constructs common to all imperative programming languages: loops are represented by their control flow, i.e., jumps between basic blocks, memory references are lowered into pointer accesses, and expressions are lowered into three-address code.

From a practical point of view, optimization passes in compilers like GCC and LLVM are implemented on this low-level intermediate representation (IR) in order to avoid duplicating analyses and optimizations for each supported language, and in order to abstract away from the specifics of each language: for instance, consider the semantics variability of loop statements in Fortran, C, C++, Java, and Ada languages, all currently compiled and optimized by the middle end of GCC. The price paid for the generality of the approach is extra compilation time spent in recognizing all the high-level loop constructs from the low-level representation. This paper describes a fast algorithm to discover, from a low-level representation, loops that can be handled and optimized by a polyhedral compiler.

IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2016, Prague, Czech Republic
In conjunction with HiPEAC 2016.

http://impact.gforge.inria.fr/impact2016

1.1 What are the boundaries of a SCoP?

Regions of code that can be handled in the polyhedral model are usually called Static Control Parts [6, 3], abbreviated as SCoPs. Usually, SCoPs may only contain regular control flow free of exceptions and other constructs that may provoke changes in control flow such as conditional expressions dependent on data (read from memory) or side effects of function calls. As the compiler cannot easily handle such constructs in the polyhedral model these statements are not integrated in a SCoP, causing a split of the region containing such difficult statements into two SCoPs.

To extend the applicability of polyhedral compilation, [2] presents techniques to represent general conditions, enlarging the limits of SCoPs to full function bodies. We will not consider these SCoP extension techniques in the current paper, as we want a fast SCoP detection suitable to be turned on by default at usual optimization levels, like "-O2", "-O3", and "-Ofast". For that, we need an algorithm that is able to quickly converge on the largest SCoPs that can profitably be transformed by a polyhedral compiler like isl [20] within a reasonable amount of compilation time. Practically, on a large number of compiled programs, we want close to zero overhead for functions without loops and for functions without any interesting loops to polyhedral compilation, and less than ten percent of the overall compilation time when optimizing loops for paths shown as hot in the profile of the compiled program.

1.2 High level overview of SCoP detection algorithms

Existing implementations of SCoP detection based on low-level representations are based on some form of representation of the control flow graph: the first implementation we did for Graphite [19] was based on the control flow graph itself and the basic-block dominators tree. Then Polly improved on this SCoP detection [10, 7] by working on an abstraction of the control-flow graph: the Single Entry Single Exit (SESE) regions tree, a representation that integrates the dominator tree with the control flow graph [12].

In order to avoid constructing the regions tree, and to help the SCoP detection algorithm converge faster on the parts of code that matter to polyhedral compilation, we use an abstraction of the control flow graph – the natural loops tree [1] – together with dominance information [16]. In terms of compilation time, both representations come at zero cost, as they are maintained intact and correct between GCC's middle-end optimization passes, lowering the compilation cost for the most numerous functions compiled by GCC: functions with no loops, or with non-contiguous isolated loops that cannot be profitably transformed in a polyhedral compilation.

1.3 Contributions of this paper

1. We present a new algorithm for SCoP detection that improves the existing techniques of extracting SCoPs from a low-level intermediate representation.

2. We provide a comparative analysis of the earlier algorithm implemented in Graphite, the SCoP detection as implemented in LLVM-Polly, and the new SCoP detection that we implemented and integrated into GCC.

3. We provide an analysis of the shape of SCoPs detected by these three algorithms in terms of the number of SCoPs detected in benchmarks and the number of loops per SCoP.

4. We present results on compilation time improvements.

2. COMPARATIVE ANALYSIS OF SCoP DETECTION ALGORITHMS

We first describe the SCoP detection algorithms implemented originally in Graphite and in Polly, and then we show how the new algorithm improves over these earlier implementations. The common point of all these algorithms is that they work on a low-level representation of the program. Analysis passes are building more synthetic representations as described in the next subsection.

2.1 Code analysis: from low-level to higher-level representations

SCoP analysis orchestrates the call to several analysis passes that will be described in this subsection. SCoP detection calls on demand each of the following analysis passes, caching the results for the polyhedral translation pass, or asking for different analysis results when the context changes, for example, when a SCoP is extended upwards, a variable that used to be considered a parameter of the SCoP could be analyzable in the context of the larger extended SCoP.

The following analysis passes are both available in LLVM and GCC. Graphite and Polly are using these analysis passes to detect SCoPs, with differences in the order, composition, and use of the passes. The main difference that will be described in detail in the paper is the use of a high level representation of natural loops and irreducible strongly connected components instead of a more raw representation of the same information under the form of dominance or dominance frontier.

Starting from a low-level representation of the program, a compiler extracts information about specific aspects of the program through program analysis:

- The control flow graph (CFG) [1] is built on top of a goto based intermediate representation: each node of the CFG represents a basic block, i.e., the largest number of statements to be executed sequentially without branches, and the edges of the CFG correspond to the jumps between basic blocks.

- The basic-block dominator (DOM) and post-dominator (Post-DOM) trees [1, 16] represent the relations between basic blocks with respect to the control flow properties of the program: a basic block A is said to dominate another basic block B when all execution paths from the beginning of the function have to pass through A before reaching B. Similarly, the post-dominator information is obtained by inverting the direction of all edges in the CFG and then asking the same question with respect to the block ending the function: A is said to post-dominate B when all paths from the end of the function have to pass through A to reach B in the edge-reversed CFG.

- The Single Entry Single Exit (SESE) regions [12] are obtained from the CFG and the DOM and Post-DOM trees by identifying contiguous sequences of basic blocks dominated and post-dominated by unique basic blocks. In LLVM the computation of SESE regions tree is based on iterated dominance frontier [16] that can be quadratic in some cases. The SESE regions may contain sub regions that have the SESE property: this inclusion relation is represented as a tree.

- Natural loops [1, 16] are detected as strongly connected components (SCC) [18] on the CFG, loop nesting and sequence are represented under the form of a tree: loop nodes are linked with *inner* loop and *next* loop relations. The function body is represented as a loop at the root of the loop tree, and its depth is zero. The depth of inner loops is one more than their parent, and sibling loops linked through *next* are at the same depth. When the CFG contains an SCC that is not reducible to a natural loop, for example two back-edges pointing back to a same basic block, all the edges and nodes of the CFG involved in that SCC are marked with a flag IRREDUCIBLE_LOOP.

  [17] contains the description of an algorithm to detect natural loops in the presence of irreducible control flow: the main objective is to detect reducible natural loops and minimize the number and span of irreducible regions, to confine the effect of irreducibility (i.e., inability to optimize loops) to small regions. To detect reducible regions nested within irreducible regions, and to detect finer irreducible regions, they use the loop nesting level information, an information that we also use in our improved SCoP detection algorithm.

- Static Single Assignment (SSA) form [4] inserts extra scalar variables such that each definition is unique. The assignments to scalar variables are either the result of expressions, or the result of phi nodes placed at control flow junctions in basic blocks that are target of two or more control flow edges. Assignments from phi nodes represent all the possible assignments considering the control flow graph. For example, the following imperative program

```
a = 0;
b = 0;
loop:
  a = a + 1;
  b = a * 2;
  if (a > 10)
    goto end;
  goto loop;
end:
c = b;
```

is translated to SSA by adding new variable names and phi nodes that merge values at control-flow junctions:

```
a_0 = 0;
b_1 = 0;
loop:
  a_2 = phi (a_0, a_4);
  b_3 = phi (b_1, b_5);
  a_4 = a_2 + 1;
  b_5 = a_4 * 2;
  if (a_4 > 10)
    goto end;
  goto loop;
end:
c_6 = phi (b_5);
```

An abstract view of the SSA representation is a declarative language [14] in which there are no assignments or imperative language constructs: it only represents the computation of scalar variables that are either the result of arithmetic expressions, or the result of two kinds of phi nodes. Loop-phi nodes define recursive expressions: the first argument of a loop-phi node defines the initial value, and the second argument defines the recursion by using self-references. For example,

```
a = loop_1-phi (0, a + 1)
b = loop_1-phi (0, 2 * a)
```

a is defined as 0 on the first iteration, and uses a self-reference expression $a + 1$ for all other iterations; b is defined to have the value 0 in the first iteration, followed by an expression that references another loop-phi node $2 * a$ for all the subsequent iterations.

Loop close-phi nodes compute the last value defined in a loop by a loop-phi node: they correspond to the min operator of partial recursive functions. For example,

```
c = loop_1-close-phi (b, a > 10)
```

c is defined as the value of b on the first iteration when a becomes greater than 10: in this particular example, the value of c can be statically evaluated to 22.

- The analysis of scalar evolutions (scev) [13] starts from the abstraction of the SSA representation described above by recognizing the evolution function of scalar variables for loop phi nodes, loop close-phi nodes, and derived scalar declarations. Loop phi nodes are declared by an initial value and an expression containing a self reference, when the self reference appears in an addition expression together with a scalar value or an invariant expression in the current loop, the scev represents a recursive function with linear or affine evolution. Loop close-phi nodes are declared as the last value computed by an expression that may variate in a loop, the scev then represents a partial recursive function. All other scalar declarations can be expressed as scevs derived from declarations of other recursive and partial recursive functions. scev analysis would provide the following expressions for the above running example:

  ```
  a = {0, +, 1}_1
  b = {0, +, 2}_1
  c = 22
  ```

  The scev of a shows that its initial value is 0 and every iteration the value linearly increments + by 1. Similarly, the initial value of b is 0 and is incremented every iteration by 2. Since c is evaluated outside the loop its value does not change.

- The analysis of the number of iterations [13] provides a scev that represents the number of times a loop is executed. The number of iterations is computed as the scev of a close-phi node, or last value, of a scev starting at zero and incremented by one at each iteration of the loop. In the running example, the number of iterations can be computed as the value of a when exiting the loop, and could be statically evaluated to 11.

  The number of iterations can also be computed by isl, as we provide the scevs of all the variables involved in each condition that we translate.

- Pointer analysis detects base and access functions for all memory locations that are accessed in the program.

- Alias analysis disambiguates the base pointers and identifies which pointers may access the same memory.

- Data reference analysis uses the results of pointer, alias, and scev analyses to determine the memory access patterns of arrays and pointers in loops. SCoP detection uses the data reference analysis to determine whether memory accesses are linear and suitable for polyhedral representation. Frequently the compiler linearizes the access functions of a multi-dimensional array into base pointer plus offset, leading to access functions that cannot be represented in the polyhedral model: for example, in the following code

  ```
  int N;
  int A[10][N];
  for (int i = 0; i < 10; i++)
    for (int j = 0; j < N; j++)
      A[i][j] = 42;
  ```

  the linear form of the array access $A[i][j]$ is $A + i * N * 4 + j * 4$. As N is a parameter and i is an induction variable, the polyhedral model cannot represent $i * N$. In order to represent such array accesses, the SCoP detection needs to verify that the access function is actually that of a multi-dimensional array, and thus it calls a delinearization pass over all memory accesses to retrieve the array multi-dimensions lost in the translation from the front-end to the middle-end.

- The delinearization analysis [9, 8] reconstructs a high-level representation of arrays under the form of Fortran subscripts. The delinearization uses all the access functions of all the data references in a loop, a region, or a function, in order to make sure all memory accesses follow the exact same pattern, i.e., the same subscript dimensions are valid in all accesses.

As we have seen, gradually, these analyzes extract from low-level constructs higher-level representations: later analyzes are based on earlier lower-level results, building up a castle out of basic bricks. All this information synthesized by the compiler allows the representation of diverse imperative programming languages into a polyhedral form [6] containing a very high-level information of loop iteration domains, memory accesses, and static and dynamic schedules.

To reduce compilation time, the SCoP detection algorithms start from the lowest-level analysis results that are commonly available and try to quickly discard parts of code that either cannot be translated in the polyhedral model or that cannot be profitably transformed. The compiler has to quickly evaluate whether a part of the code is amenable to translation in the polyhedral model before spending time in computing higher-level costly information. For that reason, information about the CFG structure, and the number of loops per function are the first checks in the SCoP detection algorithms, followed by a linear walk over all the statements of the region of code to gather more costly information.

2.2 Former SCoP detection in Graphite

The first Graphite SCoP detection algorithm was implemented on a very low-level representation of CFG and DOM [19]. These representations were too restrictive for the scev analysis to be able to determine the loops to be considered as variant and the loops that have to be considered as invariant, in which the scev analysis should not analyze and instantiate further scalar variables in order to consider them as parameters [13]. This resulted in a very restrictive limitation of SCoPs that had to have one full loop fully contained in the SCoP: for instance a sequence of two loops with no surrounding loop would not be represented as a SCoP and the SCoP detection would split these two loops into two distinct SCoPs. This limitation has been removed in GCC 6.0 by the improvements to the SCoP detection described later in this paper.

2.3 Polly SCoP detection on SESE regions

Polly's SCoP detection is based on an analysis of SESE regions [10]. The discovery of all the regions in a function may be expensive, especially when the number of basic blocks in its CFG is very large. In the current LLVM implementation of SESE region discovery, the use of dominance frontiers may have quadratic behavior in some cases. The algorithm described in this paper may help in reducing the cost of the SESE region analysis by replacing the use of dominance frontiers with a simpler check based on the natural loops properties of depth levels and regions of the CFG corresponding to irreducible loops, see Figure-3 in Section-3.

2.4 Why maximal SCoPs are not very useful

A maximal SCoP is the largest region satisfying all the properties of a SCoP and which cannot be further extended. This concept is currently used in Polly [10] and was used in the previous SCoP detection of Graphite [19].

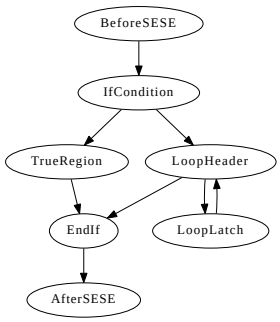

Figure 1: Maximal SCoP bounded by IfCondition and EndIf

Since the main focus in the polyhedral model is the transformation of loops, all code translated in the polyhedral model that is not part of a loop, or part of a sequence of loops, only constitutes an overhead in compilation time. Finding and representing a maximal SCoP is not necessary as it is possible to analyze all the surrounding conditions and extract constraints on the parameters while translating the IR to the polyhedral representation (after SCoP detection).

To illustrate our point, consider the maximal SCoP in Figure-1: suppose that the region from IfCondition to EndIf satisfies the properties of a SCoP, and contains a single loop composed of a LoopHeader and a LoopLatch. All the other blocks of a maximal SCoP IfCondition, TrueRegion, and EndIf add extra overhead to the polyhedral compilation without real benefit in terms of adding opportunities for loop optimizations. Having a SCoP only containing the loop would prove more beneficial in terms of optimizations and overall compilation time. The reader could argue that the guard expression in IfCondition may bring valuable context information on the domain parameters, which in turn can have a significant impact on the efficiency of the generated code. This is correct, however the compiler does not need to extend the SCoP to contain the IfCondition to be able to extract the same information about the parameters: indeed, the constraints on parameters are extracted from outer conditions, variable types, and everything else the compiler can reason about the parameter variables defined outside the SCoP.

The greedy approach of detecting maximal SCoPs is not necessarily optimal in terms of compilation time, as well as in terms of performance achieved: detecting smaller SCoPs may prove beneficial to compilation time, as well as narrowing the choices of possible optimizations that would again lead to compilation time improvements.

We do not yet have a heuristic on how to limit the SCoPs to only those regions containing loops that can be profitably transformed by a polyhedral compilation, leading to performance improvements. Our implementation of the SCoP detection still tries to maximize the size of SCoPs, with the exception that it will not detect regions of code without loops. We think that SCoP detection is the right place to implement heuristics based on the shape of the code to avoid the high cost of the translation into the polyhedral representation.

3. A NEW FASTER SCoP DETECTION

The new algorithm for SCoP detection works by induction

```
// sese: { edge entry, edge exit }
// bb: basic block

// Recurse on the loop.inner.
sese build_scop_depth (sese s1, loop l):
  s1 = build_scop_depth (s1, l.inner)
  sese s2 = merge_sese (s1, get_sese (l))
  if (s2 is an invalid scop)
    {
      // s1 might be a valid scop, so return it
      // and start analyzing from the adjacent loop.
      build_scop_depth (invalid_sese, l.next)
      return s1
    }
  if (l is an invalid scop in s2)
    return build_scop_depth (invalid_sese, l.next)
  return build_scop_breadth (s2, l)

// Recurse on loop.next.
sese build_scop_breadth (sese s1, loop l):
  sese s2 = build_scop_depth (invalid_sese, l.next)
  if (s2 is an invalid scop)
    {
      if (s1 is a valid scop)
        add_scop (s1)
      return s1
    }
  sese combined = merge_sese (s1, s2)
  if (combined is a valid scop)
    s1 = combined
  else
    add_scop (s2)
  if (s1 is a valid scop)
    add_scop (s1)
  return s1
```

Figure 2: Induction on the structure of natural loops

on the structure of the tree of natural loops as described in Section-3.1 and listed in Figure-2. The traversal of the loop tree tries to enlarge a region by attaching adjacent or outer valid regions as described in Section-3.3 and listed in Figure-3. Section-2.1 has a brief introduction to the notions and properties used in this section; we refer our readers to [16] for an in-depth discussion. The algorithm assumes the following preconditions during the SCoP detection:

- The natural loops tree represents a reducible CFG: all edges and basic blocks in an irreducible SCC are tagged with an IRREDUCIBLE_LOOP flag.
- Dominance information is available.
- Mechanism to compute the evolution of scalars in a region exists.

3.1 Induction on the structure of natural loops

The traversal of the natural loops tree starts at a loop-nest at depth one (the loop at depth zero is the function body). The code inside the loop and all its nested loops are analyzed recursively for validity, as described in Section-3.2 and function `build_scop_depth` in Figure-2. Once all the validity constraints are satisfied, the loop-nest becomes a valid SCoP and it is saved in a set of already found SCoPs.

```
sese merge_sese (sese a, sese b):
  // ncd: the nearest common dominator
  bb dom = ncd (a.entry, b.entry)
  // ncpd: the nearest common post-dominator
  bb pdom = ncpd (a.exit, b.exit)

  edge entry = nearest_dom_with_single_entry (dom)
  if (entry not found) return invalid_sese
  edge exit = nearest_pdom_with_single_exit (pdom)
  if (exit not found) return invalid_sese

  // entry and exit should be in the same loop,
  // and hence in the same sese.
  if (loop_depth (entry.src.loop_father) !=
      loop_depth (exit.dest.loop_father))
    return invalid_sese

  // edge should belong to reducible loop.
  if (entry.flag == EDGE_IRREDUCIBLE_LOOP
      or exit.flag == EDGE_IRREDUCIBLE_LOOP)
    return invalid_sese

  sese combined = new_sese (entry, exit)
  if (entry does not dominate exit
      or exit does not post-dominate entry
      or combined is an invalid scop)
    return invalid_sese
  return combined
```

Figure 3: Merging two SESE regions

```
edge nearest_dom_with_single_entry (bb b):
  if (b has 1 predecessor edge e)
    return e
  if (b has 2 predecessor edges e1 and e2)
    { // Check for a back-edge
      if (e1.src dominates e2.src) return e1
      if (e2.src dominates e1.src) return e2
    }
  b = get_immediate_dominator (b)
  return nearest_dom_with_single_entry (b)

edge nearest_pdom_with_single_exit (bb b):
  if (b has 1 successor edge e)
    return e
  if (b has 2 successor edges e1 and e2)
    { // Check for a back-edge
      if (e1.dest post dominates e2.dest)
        return e1
      if (e2.dest post dominates e1.dest)
        return e2
    }
  b = get_immediate_post_dominator (b)
  return nearest_pdom_with_single_exit (b)
```

Figure 4: Recursive computation of a single entry dominator and a single exit post-dominator

While adding a new SCoP to the set of detected SCoPs, we first remove any SCoP which either is a sub-SCoP (completely surrounded) or intersects (partially overlaps), with the new one. The SCoP which only intersects with the new one is completely lost and we do not try to recover the non-intersecting portion for now. We would like to extend this in future. The way algorithm runs, from bottom up for each loop nest, any new SCoP to be added cannot be subsumed by an existing SCoP in the set. This way, the SCoPs maintained in the set are mutually exclusive w.r.t. the regions they span i.e., no SCoP intersects with another in the set of detected SCoPs. It may be noted that the return value of `build_scop_depth` in line 10 has not been captured. This is because the algorithm goes two ways, first part s1 is to be returned while the algorithm continues from the next loop at the same depth.

After a valid loop is found, the algorithm analyzes the next loop (see `build_scop_breadth` in Figure-2), a loop at same depth and immediate sibling of the loop just analyzed. If an adjacent loop is found to be a valid SCoP, we try to merge both loop nests as described in Section-3.3 and Figure-3. If a combined SESE has been found, which subsumes both SCoPs, it is analyzed for validity. Even if we have already analyzed the statements in the combined SESE in their respective sub-SCoPs, we need to re-analyze them because the scalar evolution of the data references change with the region of the program under analysis. If the combined SESE represents a valid SCoP, then it is saved, after removing any intersecting or sub-SCoPs, and further analysis is continued to extend the SCoP again. If no such SESE could be found, the algorithm keeps them as two separate SCoPs, and continues by trying to extend the second SCoP.

With the new approach it is faster to discard many invalid loop-nests early. The algorithm analyzes statements which matter most by starting the SCoP detection from a loop-tree node (CFG node which begins from a loop header). This allows discarding unrepresentable loops early in the SCoP-detection process, thereby discarding SESE region surrounding them. This way, the number of instructions to be analyzed for validity reduces to a minimal set. We start by analyzing those statements which are inside a loop, because validity of those statements is necessary for the validity of loop. The statements outside the loop nest can be excluded from the SESE if they are not valid. Since this algorithm starts from the loop header, it excludes statements before the first, and after the last loop in an SESE. Also, regions without loops are excluded if they are not surrounded by loops. SCoPs thus detected are not maximal, in contrast with the example and discussion in Section-2.4.

3.2 When is an SESE region a valid SCoP?

An SESE region is regarded as a valid SCoP when it satisfies the following conditions:

1. The entry basic block should have only one predecessor and the exit basic block should have only one successor (i.e., an SESE).
2. The entry should dominate the exit.
3. The exit should post-dominate the entry.
4. All the loops in the SESE should have single exits.
5. The scalar evolution of all memory accesses and conditional expressions should be affine.
6. All the statements inside the region should be representable in the polyhedral model. For example, labels, pure function calls, assignments and comparison operations on integer types are allowed.
7. The induction variables of all the loops should be of (or convertible to) signed integer type because isl might generate negative values in the optimized expressions which would have to be code-generated.

3.3 Merging SCoPs

We compose a larger SESE by merging two smaller SCoPs. In order to merge two SESEs, as described in Figure-3, to form a new SESE, we search the nearest common dominator *dom* and the nearest common post-dominator *pdom* to form a new region. After that we find the nearest dominator of *dom* with single entry because we want to build an SESE. We iterate on the dominator tree until we find such basic block, as described in Figure-4. If any of the dominators has two predecessors but one of them is a back edge, then that basic block also qualifies as a dominator with single entry. Similarly, we find nearest post-dominator of *pdom* with single exit. For this, we iterate on the post-dominator tree until we find such basic block. If any of the post-dominators has two successors but one of them is a back edge, then that basic block also qualifies as a post-dominator with single exit.

After such entry and exit edges have been found, we check whether the entry basic block of the region, which is the destination of the entry edge *entry.dest*, dominates the exit basic block of the region, which is the source basic block of the exit edge *exit.src*. Similarly, *exit.src* should post-dominate the *entry.dest*. Also, the bounding basic blocks – source basic block of the entry edge and the destination basic block of exit edge – should belong to the same loop depth. It is possible to continue extending the region by finding edges satisfying both these conditions, although for now the algorithm chooses to bail out. We would like to extend this functionality in the future.

If all the previous constraints are satisfied, the algorithm returns a larger SESE which subsumes both the SCoPs, otherwise it returns an invalid SESE to inform that the merge was unsuccessful.

3.4 Analysis of the SCoP detection algorithms

The new algorithm is linear in the number of loops as it iterates on the tree of natural loops. The number of calls to the dominators and post-dominators is also linear in the number of loops.

The previous implementation of SCoP detection in Graphite was linear in the number of CFG edges as it discovers regions by walking on the CFG. In the implementation of Polly, the use of iterated dominance frontiers to build the SESE regions tree may lead to quadratic behavior in some cases [16]. This can be expensive when the function body is large, specially with aggressive inlining. The reader could argue that the quadratic behavior of Polly's SCoP detection will not occur in practice, and could only happen on pathological constructs [5] (Figures 1 and 2.) As Polly requires the analysis of dominance frontiers on all compiled functions, its SCoP detection is exposed to all the pathological constructs that could trigger the quadratic behavior.

The complexity of the validating function is still a point that could be improved in all the SCoP detection algorithms:

every statement of the SESE has to be validated. When a SCoP is extended upwards, either including in the larger region sequential loops, or going from an inner loop to an outer loop, the scev instantiation point changes, and thus the scevs of inner or lower regions become invalid under the new instantiation point, and have to be reanalyzed.

The new SCoP detection algorithm helps analyze fewer statements in case of an invalid SCoP because it focuses on the structure of the natural loops first rather than the validity of each statement.

4. EXPERIMENTAL RESULTS

To compare against the existing SCoP detection algorithms, we set up two experiments: first we look at a static metric consisting in counting the number of SCoPs and the number of loops in those SCoPs discovered on a set of benchmarks known to contain loops that benefit from polyhedral compilation. Then we evaluate how much time the compiler spends on trying to find SCoPs containing meaningful loop nests to be optimized in the polyhedral compilation on a large code-base.

4.1 SCoP metrics on Polybench

To validate the impact of our changes to the Graphite framework by replacing the old SCoP detection algorithm with the new one, we evaluate the number of SCoPs and number of loops per SCoP discovered on the Polybench [15] in Table-1. We also provide the same metrics for Polly, and we intentionally do not provide conclusions based on these metrics because a fair comparison on these metrics is difficult: the pass ordering and level in the pass pipeline at which Graphite and Polly apply are very different, and so the shapes of the CFG and natural loops tree on which the SCoP detection applies are radically different.

Metric	New	Old	Polly
SCoPs	34	189	30
Max loops/SCoP	17	8	11
Min loops/SCoP	2	1	2
SCoPs with min loops/SCoP	7	109	3
Loops in SCoPs	207	316	155
Loops/SCoP	6.09	2.59	5.17

Table 1: SCoP metrics on Polybench.

There were no regressions in Graphite while moving from the old to the new SCoP detection because the difference: $316 - 207 = 109$, corresponds to SCoPs with only one loop. The improved algorithm for SCoP detection does allow for the discovery of SCoPs with single loops, which could be profitably transformed as the polyhedral model could expose parallelism and vectorization opportunities. In our implementation we deliberately have chosen to discards all SCoPs with less than 2 loops, a choice we will revisit when we will have a reason to enable single loop SCoPs.

We also see that the new algorithm discovers larger SCoPs on an average, i.e., from 2.59 to 6.09 loops per SCoP. This is because, the new algorithm allows SESE without a surrounding loop to be a SCoP so that two adjacent outermost loops can be a SCoP, which was not possible with the old algorithm (Section-2.2). Now the largest SCoP discovered has

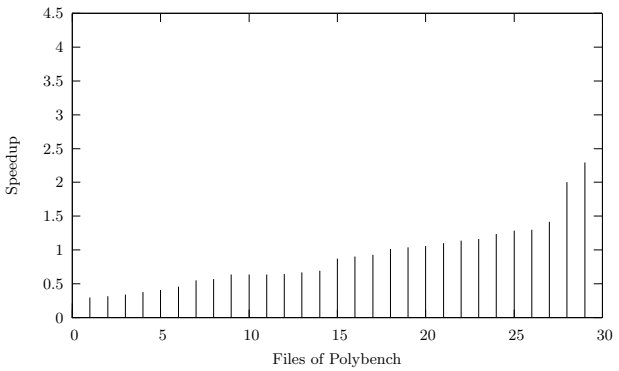

Figure 5: Speedup of improved SCoP detection on Polybench.

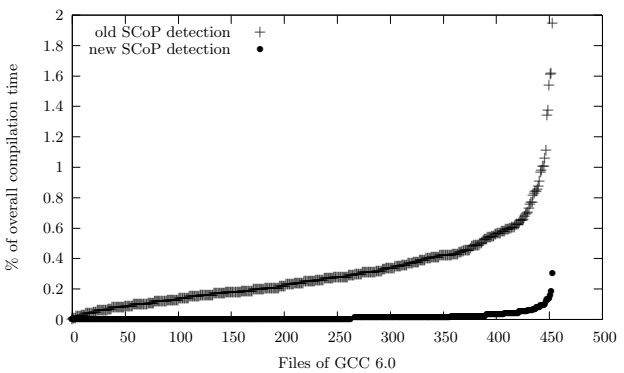

Figure 6: Overhead of the new and old SCoP detection on the overall compilation time when compiling the code of GCC 6.0.

17 loops whereas, with the old SCoP detection the largest SCoP only contained 8 loops.

4.2 Evaluation of compilation time overhead

The current infrastructure to measure the compilation time in GCC isn't precise enough to benchmark portions of passes which are not dominating the overall compilation time. On a recent x86_64 machine, the overall compilation time spent in Graphite was a fraction of a second for each benchmark in Polybench [15].

We used Valgrind to get more precise data on the number of instructions executed by the SCoP detection algorithms. The command used is:

```
$ CFLAGS="-Ofast -fgraphite-identity"
$ valgrind --dsymutil=yes --tool=callgrind \
    --dump-instr=yes cc1plus $CFLAGS $file
$ callgrind_annotate --threshold=100 \
    --inclusive=yes callgrind.out
```

The output of callgrind_annotate lists all the functions in the execution of the program together with a count of the number of instructions executed by the CPU. The option "–inclusive=yes" allows us to gather the total number of instructions executed in a function and all the functions called from it. We thus report the number of instructions of the top-level functions of the old and new SCoP detectors, respectively build_scops and build_scop_depth.

Table-2 presents the overall number of instructions executed in the new and old SCoP detection on several benchmarks: Polybench [15], a large C++ application Tramp3d-v4 [11], and all the source files of GCC 6.0. Columns New and Old report the number of instructions executed by the old and new SCoP detectors of GCC; Speedup reports the speedup between Old and New; Main reports the cumulative number of instructions executed by the main function of the GCC cc1plus compiler; Old % and New % report the overall compilation time overhead of the old and new SCoP detection. On large applications, the speedup over the old algorithm for SCoP detection is very important: on Tramp3d-v4, the old SCoP detection accounted for 7.0%, vs. 0.3% in the new SCoP detection. On Polybench there is some slowdown corresponding to the detection of larger SCoPs: every step in enlarging the SCoP may trigger extra computations for scevs. The slowdowns only correspond to benchmarks with loops that can be optimized by a polyhedral compilation.

Figure-5 reports the speedup in SCoP detection when compiling the 32 files of Polybench, and Figure-6 presents the ratio of how many instructions are used for the original and improved SCoP detection implementations against the overall number of instructions executed for the compilation of the 578 files of the source code of GCC 6.0.

It may appear to the reader that the initial SCoP detection had low enough overhead for industrial use, i.e., less than 2% of the total compilation time. However, when looking at Table-2, we see that cumulatively over the large codebase of GCC, the improvement of the new SCoP detection results in an order of magnitude reduction: $1.5e^{10}$ vs. $6.7e^{8}$ number of instructions used for SCoP detection. Essentially, in order to enable polyhedral optimization by default in an industrial compiler, the SCoP detection should be as neutral as possible, as it is the only part of a polyhedral compiler that has to scan the complete program.

Benchmark	Old	New	Speedup	Main	Old %	New %
Polybench	$3.3e^{8}$	$4.8e^{8}$	0.7	$2.5e^{10}$	1.4	1.9
Tramp3d-v4	$1.8e^{9}$	$6.2e^{8}$	2.8	$1.9e^{11}$	7.0	0.3
GCC 6.0	$1.5e^{10}$	$6.7e^{8}$	22.6	$6.1e^{12}$	0.24	0.01

Table 2: Overall number of instructions spent in SCoP detection.

5. CONCLUSION AND FUTURE WORK

We have shown that our new algorithm of SCoP detection is faster in terms of compilation time and that it detects larger SCoPs than the previous implementation in Graphite. This stems from the fact that operating on a higher-level program representation allows the algorithm to converge faster on regions that cannot be represented in the polyhedral model.

The improvements in compilation time assure that enabling the polyhedral optimizations by default at common optimization levels will not slow down the compilation time for the majority of programs compiled by an industrial compiler: i.e., programs with no loops, or with sparse loops that should not be translated into the polyhedral model.

While the new algorithm focuses only on detecting SCoPs with relevant loops, it still tries to maximize the size of the SCoP. This may lead to increased polyhedral compilation time. We think that the SCoP detection should take a more active role in driving the polyhedral optimizations by analyzing the shape of the SCoPs and selecting appropriate polyhedral optimizations.

Another way to improve the compilation time of an industrial polyhedral compiler, that is not discussed in this paper, is by using the profile information collected during an earlier execution of the program and made available to the compiler during subsequent compilation: one could use the profile information to tune the SCoP detection algorithm to focus only on the hot paths of the program. We are also investigating techniques to tune the amount of compilation time allocated to the polyhedral optimizer (isl [20] in case of gcc).

6. ACKNOWLEDGEMENTS

We would like to thank Samsung Austin R&D Center for supporting our work on improving Graphite and Polly. We would also like to thank Tobias Grosser, Johannes Doerfert, and Michael Kruse for discussions on improving the existing SCoP detection in Polly.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison wesley, 1986.

[2] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[5] R. K. Cytron and J. Ferrante. Efficiently computing φ-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):487–506, 1995.

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

[7] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[8] T. Grosser, S. Pop, L.-N. Pouchet, and P. Sadayappan. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 351–360. ACM, 2015.

[9] T. Grosser, S. Pop, J. Ramanujam, and P. Sadayappan. On recovering multi-dimensional arrays in polly. In *Proceedings of the Fifth International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2015.

[10] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[11] R. Guenther. Tramp3d-v4: a template-intensive numerical program based on freepooma. *URL: http://www.suse.de/~rguenther/tramp3d/*.

[12] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[13] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers*, pages 218–232. Springer, 2005.

[14] S. Pop, P. Jouvelot, and G. A. Silber. In and out of ssa: A denotational specification. In *Workshop Static Single-Assignment Form Seminar,*, 2007.

[15] L.-N. Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://sourceforge.net/projects/polybench*, version 4.1.

[16] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[17] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using dj graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, 1996.

[18] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

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

[20] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.