DeLICM
Undoing SSA Transformations for Polly

Michael Kruse[1,2] and Tobias Grosser[3]

[1] École Normale Supérieur, 45 rue d'Ulm, F-75005 Paris
[2] Inria Paris, 2 rue Simone Iff, F-75589 Paris Cedex 12
[3] ETH Zürich, Rämistrasse 101, CH-8092 Zürich

Abstract. Increasing data movement costs motivate the integration of advanced loop optimization frameworks in the standard flow (-O3) of production compilers. While high-level loop optimizers are effective when directly applied on unmodified hand-crafted code, they have close to zero impact when integrated in a standard compilation flow. Scalar dependencies (dependencies carried over a single memory location) are the main obstacle preventing effective optimization. We present DeLICM, a set of transformations which, backed by a polyhedral value analysis, eliminate problematic scalar dependences by 1) relocating scalar memory references to unused array locations and by 2) forwarding computations that otherwise cause scalar dependences. Our experiments show that DeLICM effectively eliminates complex dependences introduced by human programmers, compiler-internal canonicalization passes, optimizing code generators, or arising as inlining artifacts – all without the need for any additional memory allocation. As a result, advanced loop optimizations can be applied without being disturbed by, or even benefiting from, canonicalizations essential for optimizing modern C++ code.

1 Introduction

Advanced high-level loop optimizations, often using polyhedral program models [1], have been shown to effectively improve data locality for computational codes from image processing [2], over iterated stencil [3], lattice-boltzmann computations [4] to sparse-matrices [5] and even provide some of the leading automatic GPU code generation approaches [6, 7].

While many polyhedral optimizers have shown successful in research, their production use is still not common because none of the established frameworks is enabled by default in a standard optimization setting (e.g., -O3). With GCC/graphite [8], IBM/XL-C [9], and LLVM/Polly [10] there are three polyhedral optimizers that work directly within a production C/C++/FORTRAN compiler and are shipped to millions of users. While all provide good performance on various benchmarks, one of the last roadblocks that prevents running optimizers such as LLVM/Polly by default is the fact that Polly is not well integrated into the standard compilation flow of LLVM, but instead runs as pre-processing pass before the standard -O3 sequence.

```
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] += A[i] * B[k];
   }
```

Listing 1: Example: sums-of-products.

```
   double a;
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0; a = A[i];
     for (int k = 0; k < K; k += 1)
S:     C[i] += a * B[k];
   }
```

Listing 2: Sums-of-products after hoisting the load of `A[i]`.

While running early in the pass pipeline makes it easy to detect specially hand-crafted loop kernels, it can also cause random performance changes and negatively impact inlining decisions. To make Polly effective early in the pass pipeline an additional set of canonicalization passes that prepare the program code for the loop analysis have to be run on all input that should potentially be optimized by Polly, which introduces random IR changes and consequently performance noise that is hard or even impossible to control. At the same time, applying loop optimizations – which often increase code size – early means that no inlining has yet been performed and consequently smaller loop regions are exposed or future inlining is prevented. Hence, running Polly early works only in controlled benchmarking situations, but is not something that can be shipped to millions of users.

The right position for a polyhedral optimizer is late in the pass pipeline, after all inlining and canonicalization has taken place. In this position, the IR can be analyzed without having to modify the IR, such that any IR changes finally introduced are intentional and known-beneficial according to the performance models of the polyhedral loop optimizer and that the inlining passes have eliminated the C++ abstraction overhead. However, when running late in the pass pipeline – after inlining and canonicalization – various simplifications are performed that introduce scalar dependencies and prevent most loop transformation (section 5).

While most research loop optimizers [11, 6] expect input code to work mostly on arrays, in practice many programs work – at least in parts – on temporary scalars. These scalars are common after inlining and canonicalisation but often also arise for various other reasons (section 1.1).

For instance, listing 1 works entirely on arrays and is easier to optimize than listings 2 to 4 due to not having scalar dependencies. This is because of the temporaries `a` and `c`. Although the data flow dependencies are the same, new false dependencies are introduced that prevent the scalars `a` or `c` from being

```
   double c;
   for (int i = 0; i < N; i += 1) {
T:   c = 0;
     for (int k = 0; k < K; k += 1)
S:     c += A[i] * B[k];
U:   C[i] = c;
   }
```

Listing 3: Sums-of-products after promotion of `C[i]` to a scalar.

```
   double c;
   for (int i = 0; i < N; i += 1) {
T:   c = 0;
     for (int k = 0; k < K; k += 1)
S:     C[i] = (c += A[i] * B[k]);
   }
```

Listing 4: Sum-of-products after partial promotion of `C[i]` to a per-iteration temporary.

overwritten before all statements using that value have been executed. For listing 3 this means that the dynamic statement instance $T(i)$, statement T at loop iteration i, cannot execute before the previous iteration $i-1$ has finished. In this case, statement instance $U(i-1)$ must execute before $T(i)$, represented by the (anti-)dependencies

$$\forall i \in \{\, 1, \ldots, N-1 \,\} : U(i-1) \to T(i) \ .$$

We call dependencies that are induced by a scalar memory location such as `a` or `c`, *scalar dependencies*. Such scalar dependencies effectively serialize the code into the order it is written, inhibiting or complicating any advanced loop optimizations including tiling, strip-mining, and parallelization. Certain specific optimizations can work well with scalar dependencies. A vectorizer can for example introduce multiple independent instances of the scalars, one for each SIMD lane. An automatic parallelizer can privatize the temporaries per thread. However, general purpose loop scheduling algorithms such as Feautrier [12, 13] or Pluto [11] do not take privatization opportunities into account. In case transformations such as loop distribution or reversal should be applied, it is not even possible to privatize variables to avoid scalar dependencies. Hence, to allow maximal scheduling freedom it is desirable to model programs with a minimal number of scalar dependencies or none at all (listing 1). As the semantics of listings 1 to 4 are identical any good compiler should be able to compile them to the same optimized program, after all loop optimizations have been applied.

In this article we present a novel memory dependence removal approach that reduces a polyhedral's compiler sensitivity on the shape and style of the input code and enables effective high-level loop optimizations as part of a standard optimization pipeline.

Our contributions are:

- A polyhedral value analysis that provides at statement instance granularity information about the liveness and the source statement of individual data values.
- Greedy Coalescing: Maps promoted scalars to array elements that are unused to handle cases such as listings 3 and 4. (section 2)
- Operand Tree Forwarding: Move instruction and memory reads to the statements where they are used to handles cases such as listing 2. (section 3)
- An evaluation showing that advanced loop optimizations are effective late in the pass pipeline. (section 5)

Our approach has been implemented in LLVM/Polly.

1.1 Sources of Scalar Dependencies

Scalar memory dependencies arise for several reasons:

Single Static Assignment (SSA) based intermediate representations (IR) used in modern compilers cause memory often to be promoted to virtual registers, like the scalar temporaries `a` or `c` from the examples. Values that are defined in conditional statements are forwarded through φ-nodes. Both are commonly modeled as scalar accesses in polyhedral optimizers.

Loop-Invariant Code Motion moves instructions that evaluate to the same value in every loop iteration, in front of the loop, and instructions whose result is only used after the loop, after the loop. In LLVM, it is performed by two passes: LICM and GVN. The former also promotes memory to virtual registers while being in the loop, which transforms listing 1 into listing 3. A part of GVN for partial redundancy elimination (PRE) pass can transform listing 1 into listing 2 or listing 4.

C++ abstraction layers allow higher-level concepts to be represented as objects where components are distributed over different functions. For instance, a class can implement a matrix subscript operation by overloading the call operator. When invoked the operator will return the content at the indexed matrix location as a scalar. To optimize across such abstraction layers, it is essential for inlining to have taken place. The heuristics of LLVM's inliner assume that some canonicalization, including SSA-conversion, have taken place and that LICM has been performed to reduce code size and potentially eliminate side-effect free loops. Hence, the inlining required to optimize C++ code will unavoidably introduce scalar dependences.

Manually applied optimizations Programmers may write code in the style of listing 3 to not rely on the compiler to perform LICM because compilers may not reliably perform it themselves. For instance, possible aliasing between array `C` and `A` may prevent this transformation. As even C++17 does not yet provide annotations (e.g., restrict) to declare absence of aliasing, such optimizations are potentially necessary.

```
1  Knowledge = computeKnowledge();
2  for (w : Writes) {
3    if (!eligibleForMapping(w))
4      continue;
5    t = getAccessRelation(w);
6    Workqueue = getInputScalars(w);
7    while (!Workqueue.empty()) {
8      s = Workqueue.pop();
9      if (!eligibleForBeingMapped(s, t))
10       continue
11     CandidateKnowledge = computeKnowledge(s);
12     Proposal = mapKnowledge(CandidateKnowledge, t);
13     if (isConflicting(Knowledge, Proposal))
14       continue;
15     for (a :  ReadsTo(s) ∪ WritesTo(s))
16       a.setAccessRelation(Target);
17     Knowledge = Knowledge ∪ Proposal;
18     for (w : WritesTo(s))
19       Workqueue = Workqueue ∪ getInputScalars(s);
20   }
21 }
```

Listing 5: Greedy scalar mapping algorithm.

Code generators which are not not meant to generate output read by humans, might be inclined to generated some form of pre-optimized code. TensorFlow XLA for instance, passes code to LLVM with pre-promoted variables in its matrix multiplication kernels [14].

2 Value Coalescing

In this section we present the idea of reusing an array element to store a scalar.

2.1 The Greedy Scalar Mapping Algorithm

After we know the rules when two variables can be coalesced, we need an algorithm that selects the memory to be coalesced. The algorithm's pseudocode is shown in listing 5. It maps the first variable it sees without backtracking after a possible mapping is found, i.e., a greedy algorithm. In the following paragraphs we explain the algorithm in detail.

The algorithm begins by collection data about the system, which we call "Knowledge" (line 1). It includes information about each memory location referenced in this system, scalars as well as array elements:

- All writes to the location.
  - The reaching definition derived from the writes.

    - The value written by the write is added to the known content the location during the reaching definition's zone.
- All reads from the location.
    - The lifetime zones derived from the reads and the reaching definitions.
    - When using SSA, the read result itself is also an identifiable value which is added to the known content of the definition zone.
- The unused locations, which is the zone when a location is not alive. An alternative method to compute this are the zone from the last read of a reaching definition to the the of the reaching definition's zone.

All of these can be represented as relational polyhedra with dimensions tagged as belonging to an array element, timepoint, statement or value. Every write potentially opens up an unused zone right before it. Some targets might be unsuitable, for instance because the store is not in a loop or the span of elements written to is only a single element (like in **??**). Such targets are skipped (lines 3–4). For a given write, we collect two properties: first, the target array element for each statement instance that is overwritten and therefore causes an unused zone before it (`Target` on line 5); second, the list of possible scalars that might be mapped to it (`getInputScalars` at line 6). The best candidate for mapping is the value that is stored by the write because it would effectively move the write to an earlier point where the value is written (like $U(i)$ in **??**). These candidates are used to initialize a queue, with the best candidate coming out first. Prioritized candidates are those that lead to a load of the map target.

Let `s` be the next candidate scalar (line 8). Similar to checking the eligibility of `w` being a mapping target, we check whether `s` is eligible to be mapped to `t` (Lines 9–10). Reasons for not being a viable scalar to be mapped include being larger than the target location, being defined before the SCoP or used after it (because these accesses cannot be changed). The exact requirements depends on the intermediate representation used by the implementation. Once `s` passes the basic checks, we compute the additional knowledge for it in case it is mapped to `t`. It consists of the lifetime for the previously unused array elements of `t` and the known content of `s`, which then becomes the known content of `t`. We call this supplemental knowledge the `Proposal` (line 11).

The proposal is then compared with the existing knowledge to see whether there are conflicts (line 13). For two knowledges A and B to be compatible (non-conflicting), the must meet following conditions for every memory location:

- The lifetime zones of A and B must be disjoint, except when their share the same known content.
- Writes in A cannot write into B's lifetime zone, except if the the written value is already B's known content.
- Writes in B cannot write into A's lifetime zone, except if the the written value is already A's known content.

A conflict is a violation against one of these rules. They ensure that the semantics of the program remain the same after carrying out the mapping of the proposal. If a conflict is found, the proposal is rejected and the next value from the queue is

```
   double a, a_S;
   for (int i = 0; i < N; i += 1) {
T:   C[i] = 0; a = A[i];
     for (int k = 0; k < K; k += 1)
S:     a_S =  A[i];
       C[i] += a_S * B[k];
   }
```

Listing 6: Operand tree forwarding applied on sum-of-products (listing 2)

tried. If there is no conflict, then the proposal is applied. All uses of s are changed to uses of `t` (line 16). The proposal is used to update the SCoP's knowledge (line 17) by uniting the lifetimes and known content. Since writes to `s` have become writes to `t`, an new unused zone ends at each write, to which which variables can be mapped to. As with the original write (line 6), we add possible mapping candidates to the work queue (line 19). The rationale is that lifetimes interesting for mapping are adjacent: When the lifetime of one variable ends, it is because it computes a new value that might be eligible for mapping. The algorithm runs until no more variable can be mapped to `t` and then continues with the next write, until all writes have been processed (line 2).

3 Operand Tree Forwarding

The mapping algorithm is only able to remove scalar dependencies if there is an unused array element, or one that stores the same value anyway. For `a` in listing 2 this does not apply. Fortunately, we can just repeat the computation of `a` in every statement it is used in. The result of such a forwarding is shown in listing 6. Thereafter the variable `a` becomes dead code. We call this *operand tree forwarding* because for a use of a scalar in a statement, it copies the entire operand tree for the scalar in order to avoid having a scalar dependency for any of the operands.

3.1 The Operand Tree Forwarding Algorithm

We consider two kinds of operations that can forwarded to another location. The first kind are operations without side-effects and whose' result depend only on its operands. In LLVM, these are called *speculatable* instructions. The second kind is a read from memory. These can be copied if the memory location has the same content at the position of the original access and the position it is copied to.

Instead of determining whether the memory location is unchanged, we can reuse the knowledge we already need to obtain for greedy mapping. Given the known content information at the timepoint of the target of the forwarding, any value required by the target statement can be reloaded. The two advantages are

```
1  bool canForwardTree(
2      Stmt Target, Operation s, Knowledge K) {
3    e = K.findElementWithSameContent(s, Target);
4    if (e)
5      return true;
6    if (s is speculatable)
7      return false;
8    for (op : operands(s))
9      if (!canForwardTree(Target, s, K))
10       return false;
11   return true;
12 }
```

Listing 7: Determine whether an operand tree is forwardable.

```
1  void doForwardTree(
2      Stmt Target, Operation s, Knowledge K) {
3    e = K.findElementWithSameContent(s, Target);
4    if (e) {
5            Target.add(new Load(e));
6            return;
7    }
8    Target.add(s);
9    for (op : operands(s)) {
10     doForwardTree(Target, s, K);
11   }
12 }
```

Listing 8: Copy an operand tree to `TargetStmt`.

that the reload does not need to happen from the same location as the original load, and the reloaded value does not even need to be a result of a load.

Listing 7 shows the pseudocode that determines whether an operand tree rooted in s whose result is used in another statement, can be forwarded to that target statement. If `canForwardTree` returns a positive result, the procedure `doForwardTree` in listing 8 can be called to carry-out the forwarding.

Both procedures are recursive functions on the operand tree's operation. They are correct on DAGs as well, but lead to duplicated subtrees. `doForwardTree` basically repeats the traversal of `canForwardTree`, but can assume that every operation can be forwarded. The method `findElementWithSameContent` (line 3) looks for an array element for each statement instance of the target that contains the result of operation `s`. If it finds such locations, the operand tree does not need to be copied but its result can be reload from these locations. The new load is added to the list of operations executed by the target statement (line 5 of `doForwardTree`). If the operation is speculatable (line 6 in `canForwardTree`), we need to also check

whether its operands are forwardable (line 9) since copying the operation to the target will require its operands to be available as well. In `doForwardTree`, the speculatable operation is added to the target's operations (line 8) and a recursive call on its operands ensures that the operands values are as well available in the target statement (line 10).

4 Implementation

Both transformations, value mapping and operand tree forwarding, have been implemented in *Polly* [10]. Polly inserts itself into the LLVM pass pipeline depending on the `-polly-position` option, as shown in fig. 1, which in the following we shorten to *early* and *late*.

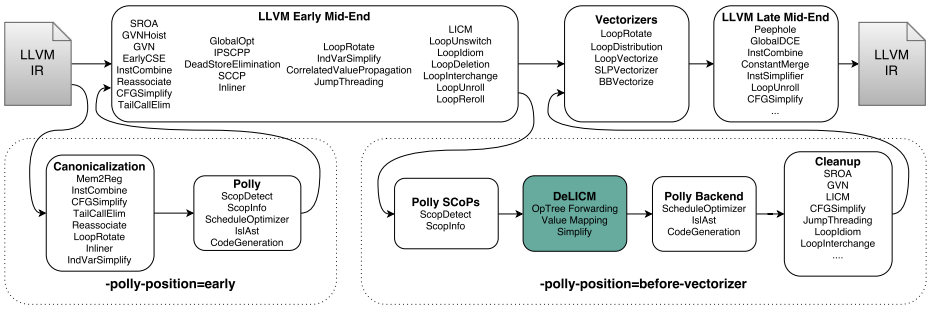

Fig. 1. Position of the Polly polyhedral optimizer in the LLVM pass pipeline.

LLVM's early mid-end is (also) meant to canonicalizethe input IR: to find a common representation for inputs that have the same semantics. Transformations should never make a follow-up analyses worse, e.g., not introduce irreducible loops or remove pointer aliasing information. Loop-Invariant Code Motion and Partial Redundancy Elimination is part of the IR's "normal form". Low-level and most target-specific transformations take place later in the pipeline.

Because scalar dependencies are destructive to polyhedral optimization, Polly's default configuration used to be early. Because Polly depends on LLVM's loop detection and its ScalarEvolution analysis, a selected set of passes are added before Polly. For functions that do not contain any loops that Polly could optimize, these are redundant because these passes run anyway in LLVM's own canonicalization phase.

In the late position, Polly is added after LLVM's own canonicalization. It first runs the operand tree forwarding followed by the greedy mapping algorithm. A simplification pass cleans up now unnecessary accesses and statements. This has many advantages:

– The IR is more canonicalized;

– The inliner has run; C++ overloaded operators are normally inlined. Hence, inlining allows Polly to see not just opaque function calls, but to optimize through C++ abstraction layers.
– Shared canonicalization passes do not run redundantly.
– No modification of the IR if Polly does not optimize anything. Even small changes can have effects on the output produced, e.g. because the instruction selection produces a different instruction. This can have unwanted and difficult to avoid effects on performance (positive as well as negative).

Polly's generated IR does not conform to LLVM's IR normal-form, so the LLVM canonicalization passes run again if Polly produced some new IR in the function. The pipeline is tuned for the current pass order (e.g. the inliner's heuristics) and used in production, therefore adjusting LLVM's canonicalization passes for the needs of Polly is not an option.

4.1 IR-Specific Algorithm Details

Because LLVM-IR is an SSA-based language, the result of operations can be identified by the operations itself. Moreover, the value can be uniquely identified within a SCoP's execution by the operation and the statement instance it is computed at. We use this for known content analysis. For instance, the value `a` produced by statement $T(i)$ in listing 2 is represented by $[T(i) \to a]$. Because `a` is loaded from the array `A`, the content of $A[i]$ at the time of the execution of $T(i)$ can also be identified as $[T(i) \to a]$.

LLVM-IR also has φ-nodes that represent different values depending on the control flow. Because the control flow is statically known within a SCoP, we can replace φ-nodes by the value they stand for. The exception is if the ϕ recursively refers to itself, in this case it is treated as a normal SSA value.

In addition to the SSA-value for a φ, Polly models a φ with an additional memory location. Predecessor statements write their value to that location such it can be read from there by the statement containing the φ. Listing 5 can map SSA- and φ-scalars alike. The difference is that φ have multiple writes, but just one read.

5 Evaluation

The code is optimized for and run on an Intel Xeon E5-2667 v3 (Haswell architecture) running at 3.20 GHz. Its cache sizes are 32 KB (L1i), 32 KB (L1d), 256 KB (L2) and 20 MB (L3). The machine has 8 cores and Hyperthreading enabled, i.e., 16 logical threads, but we only evaluate single-thread performance. The hardware details are mostly relevant for the performance tests, but some early LLVM passes make use of target-specific information as well. Hence, hardware details can influence which transformations are carried out. DeLICM itself does not make use of hardware architecture information. The compiler used for all evaluations is based on clang 6.0.0 (SVN revision 312874) using the optimization level `-O3 -march=native -ffast-math` for all tests. All evaluation is done

on Polybench/C 4.2.1 beta [15]. Polybench is a collection of 30 benchmarks with kernels typical in scientific computing. Polybench/C benchmarks are written with polyhedral source-to-source compilers in mind with the consequence that it avoids scalar variables since they are known to not work well in the polyhedral model.

Instead, we let the LLVM scalar optimizers (mostly LICM and GVN) introduce the scalar dependencies by running Polly late in LLVM's pass pipeline. Any dependency introduced by these are, by definition, avoidable. This gives the opportunity to evaluate what percentage of scalar dependencies DeLICM can remove, and how well Polly can handle LLVM's more canonicalized IR left by more passes.

Unfortunately, the combination of the LLVM passes LoopRotate and JumpThreading can destroy canonicalized loops. LoopRotate inserts a special path if a loop body is not executed at all, including nested loops. JumpThreading makes that path skip the loop exit check. This results in the outer loop having two latches, which ScalarEvolution cannot analyze. We modified the Polybench source to avoid this problem, which is a shortcoming of LLVM's canonicalization and not in the realm of Polly. We plan to write and upstream a fix to LLVM.

In order to allow LLVM's scalar optimizers to work better, we enable Polybench's use of the C99 `restrict` keyword (`-DPOLYBENCH_USE_RESTRICT`). This tells the compiler that arrays do not alias each other which allows for more code movement. For all tests, we use Polybench's large dataset (`-DEXTRALARGE_DATASET`), which sets the problem size such that the computation requires about 120MB of memory.

To show the effect of C++ and inlining, we translated four of the benchmarks to C++ using uBLAS [16]: covariance, correlation, gemm and 2mm. The former two use the original algorithm but instead of an array, they use uBLAS's vector and matrix types. The latter two use uBLAS own implementation of the matrix-matrix product. uBLAS is a header-only library with extensive use of expression templates and has no target-specific optimizations.

5.1 Performance Evaluation

The runtime speedups are shown in fig. 2 on a logarithmic scale. The speedups are relative to the execution time of the benchmark compiled with clang -O3 without Polly. Each execution time is the median of 5 runs. Each run of Polybench takes in the range of 1 hour to complete. The benchmarks gemver, symm, mvt, cholesky, durbin, gramschmidt, lu, ludcmp, deriche, floyd-warshall, nussinov, heat-3d and jacobi-2d are not shown because neither the pipeline position nor DeLICM have a significant effect on their execution speed. This does not mean that DeLICM was failed, but that the optimization chosen by Polly does not change the performance.

Generally, executing Polly at the late position causes the performance to fall back to the performance without Polly. Most of the time this is because Polly does not even generate any code because of the scalar dependencies. Not

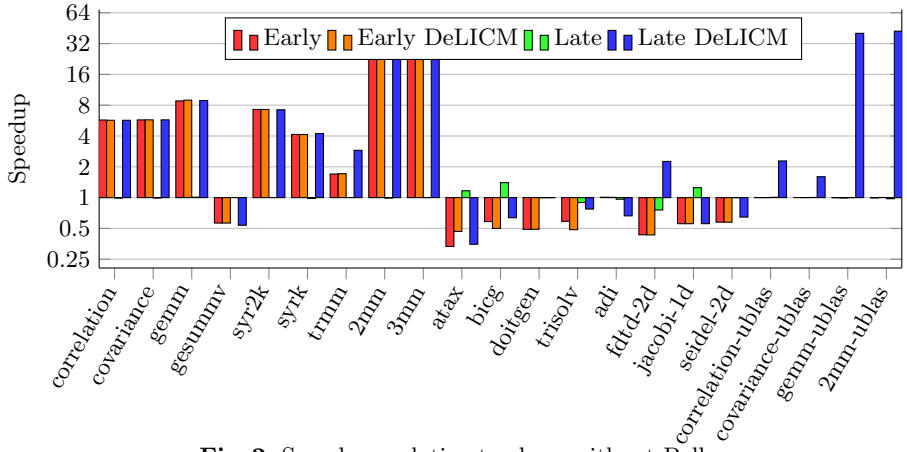

Fig. 2. Speedups relative to clang without Polly.

in all cases there is a performance gain, we also get slowdowns. The choice of optimization is not the topic of this work, but to enable such optimizations. We gain the highest speed-ups with the matrix-multiplication family (gemm, 2mm, 3mm) due to a special optimization path for it. Applying DeLICM at the beginning of the LLVM pipeline, without LICM and GVN, has little effect. Running it at the late setting almost always restores the original performance. The only benchmark where an original performance-gain could not be recovered is doitgen. In trisolv and adi some transformation could be applied, but which cause a regression

For some benchmarks, we even got a (higher) speed-up where there was none before: trmm and fdtd-2d. Early in the pipeline, the C++ benchmarks could not be optimized at all, but late in the pipeline can get optimized just as well. Covariance and correlation do not reach the performance of the original Polybench programs because of the aforementioned LoopRotation/PRE interaction. The uBLAS matrix-multiplication benchmarks are not vectorized by clang without Polly , which leads to an even higher speedup when it is vectorized by Polly.

6 Related Work

We can categorize the work in this area into two categories: memory duplication and ignoring non-flow dependencies.

The first category provides additional memory such that there are fewer conflicts between unrelated computations. Taken to the most extreme, every statement instance writes to its dedicated memory [17]. Like in SSA, every element is only written once and therefore this mode is also called Dynamic Single Assignment (DSA) [18]. This eliminates all false dependencies at the cost of greatly increasing of the memory requirement. Refinements of this technique only duplicate a private copy per thread [19]. This can solve the problem for parallelism

but does not enable inner-thread scheduling such as tiling. [20] proposes privatization on demand, when an optimization finds restricting dependencies. There is the possibility to contract array space after scheduling [21–26]. However, the re-scheduled program may inherently require more memory than the source program, especially if the scheduler is unaware that its decision may increase the memory footprint.

The second strategy is to remove non-flow dependencies when it can be shown they are not important to the correctness of the transformed program. Which dependencies can be removed depends on the scheduling transformation to be performed. A technique called *variable liberalization* [27] allows loop-fusion. Tiling and parallelism can be recovered using *live-range reordering* [28, 29].

Both come with the disadvantage that they are only compatible for the specific optimization they were designed for. In case of *live-range reordering*, the PluTo algorithm is still used with the scalar dependencies to determine bands, sets of perfectly nested loops. Live-ranges are then used to determine whether these bands are tileable or even parallel. They are if the life ranges are restricted to the band's body. This means that with this algorithm Listing 1 has a tileable band of two loops while for listings 2 to 4 the two loops that cannot be combined into a band. Still, the outer loops would be considered parallel .

A conflict set as presented in [25, 26] can also be useful to determine whether a scalar is conflicting with an array (listing 5 line 13). For the purpose of DeLICM we additionally need to analyze the stored content and one participant of the conflict set is always a scalar such that a full-array conflict set is not needed.

7 Conclusion

This paper presents two algorithms to reduce the number of scalar dependencies that are typically the result of scalar mid-end optimizations such as Loop-Invariant Code Motion, but also appear in hand-written source code. We use "DeLICM" as an umbrella name for both algorithms. The first algorithm handles the more complicated case of register promotion, where an array element is represented as scalars during a loop execution. The approach is to map the scalars back to the array element they were promoted from. Because the information which scalars were, if at all, promoted from which memory, and finding an optimal solution like register allocation would be NP-complete, we employ a greedy strategy which works well in practice. The second algorithm implements operand tree forwarding. It copies side-effect-free operations to the statements their result is needed or reloads values from array elements that are known to contain the sought value. We implemented both algorithms in Polly, a polyhedral optimizer for the LLVM intermediate representation. The experiments have shown that we can remove almost all avoidable scalar dependencies. Thanks to this effort, it also becomes worthwhile to execute Polly after LLVM's IR normalization and inlining, allowing, for instance, polyhedral optimization of C++ code.

References

1. Verdoolaege, S.: Presburger formulas and polyhedral compilation. (2016)
2. Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. In: ACM SIGPLAN Notices. Volume 50., ACM (2015) 429–443
3. Bandishti, V., Pananilath, I., Bondhugula, U.: Tiling stencil computations to maximize parallelism. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, IEEE (2012) 1–11
4. Pananilath, I., Acharya, A., Vasista, V., Bondhugula, U.: An optimizing code generator for a class of lattice-boltzmann computations. ACM Transactions on Architecture and Code Optimization (TACO) **12**(2) (2015) 14
5. Venkat, A., Hall, M., Strout, M.: Loop and data transformations for sparse matrix code. In: ACM SIGPLAN Notices. Volume 50., ACM (2015) 521–532
6. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gomez, J., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for cuda. ACM Transactions on Architecture and Code Optimization (TACO) **9**(4) (2013) 54
7. Baskaran, M., Ramanujam, J., Sadayappan, P.: Automatic c-to-cuda code generation for affine programs. In: Compiler Construction, Springer (2010) 244–263
8. Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J., Upadrasta, R.: Graphite two years after: First lessons learned from real-world polyhedral compilation. In: GCC Research Opportunities Workshop (GROW'10). (2010)
9. Bondhugula, U., Gunluk, O., Dash, S., Renganarayanan, L.: 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, ACM (2010) 343–352
10. Grosser, T., Groesslinger, A., Lengauer, C.: Polly – performing polyhedral optimizations on a low-level intermediate representation. Parallel Processing Letters **22**(04) (2012) 1250010
11. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. Volume 43., New York, NY, USA, ACM (June 2008) 101–113
12. Feautrier, P.: Some efficient solutions to the affine scheduling problem – part i. one-dimensional time. **21**(6) (October 1992) 313–347
13. Feautrier, P.: Some efficient solutions to the affine scheduling problem – part ii. multidimensional time. **21**(6) (December 1992) 389–420
14. Agarwal, A.: Enable polyhedral optimizations in xla through llvm/polly. Google Summer of Code 2017 final report (2017)
15. Pouchet, L.N., Yuki, T.: Polybench 4.2.1 beta (2016)
16. Koch, M., Walter, J.: Boost ublas
17. Feautrier, P.: Array expansion. In: ACM International Conference on Supercomputing 25th Anniversary Volume, New York, NY, USA, ACM (2014) 99–111
18. Vanbroekhoven, P., Janssens, G., Bruynooghe, M., Catthoor, F.: Transformation to dynamic single assignment using a simple data flow analysis. In Yi, K., ed.: Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings, Berlin, Heidelberg, Springer (2005) 330–346
19. Li, Z.: Array privatization: A loop transformation for parallel execution. Technical Report 9226, Univ. of Minnesota (1992)

20. Trifunovic, K., Cohen, A., Razya, L., Li, F.: Elimination of memory-based dependences for loop-nest optimization and parallelization. In: GROW 2011 : 3rd Workshop on GCC Research Opportunities, Chamonix, France (June 2011)
21. Wilde, D.K., Rajopadhye, S.: Allocating memory arrays for polyhedra. Research Report RR-2059, INRIA (1993)
22. Lefebvre, V., Feautrier, P.: Automatic storage management for parallel programs. Parallel Computing **24**(3) (1998) 649 – 671
23. Quilleré, F., Rajopadhye, S.: Optimizing memory usage in the polyhedral model. ACM Trans. Program. Lang. Syst. **22**(5) (September 2000) 773–815
24. Bhaskaracharya, S.G., Bondhugula, U., Cohen, A.: Automatic storage optimization for arrays. ACM Trans. Program. Lang. Syst. **38**(3) (April 2016) 11:1–11:23
25. Darte, A., Isoard, A., Yuki, T.: Extended lattice-based memory allocation. In: Proceedings of the 25th International Conference on Compiler Construction. CC 2016, New York, NY, USA, ACM (2016) 218–228
26. Bhaskaracharya, S.G., Bondhugula, U., Cohen, A.: Smo: An integrated approach to intra-array and inter-array storage optimization. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '16, New York, NY, USA, ACM (2016) 526–538
27. Mehta, S., Yew, P.C.: Variable liberalization. ACM Trans. Archit. Code Optim. **13**(3) (August 2016) 23:1–23:25
28. Baghdadi, R., Cohen, A., Verdoolaege, S., Trifunović, K.: Improved loop tiling based on the removal of spurious false dependences. ACM Trans. Archit. Code Optim. **9**(4) (January 2013) 52:1–52:26
29. Verdoolaege, S., Cohen, A.: Live range reordering. In: 6th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC). Prague, Czech Republic. (2016)