DeLICM:
Scalar Dependence Removal at Zero Memory Cost

Michael Kruse
Inria
Paris, France
michael.kruse@inria.fr

Tobias Grosser
ETH Zürich
Zürich, Switzerland
tobias.grosser@inf.ethz.ch

Abstract

Increasing data movement costs motivate the integration of polyhedral loop optimizers in the standard flow (-O3) of production compilers. While polyhedral optimizers have been shown to be effective when applied as source-to-source transformation, the single static assignment form used in modern compiler mid-ends makes such optimizers less effective. 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 dependencies introduced by compiler-internal canonicalization passes, human programmers, optimizing code generators, or inlining – without the need for any additional memory allocation. As a result, polyhedral loop optimizations can be better integrated into compiler pass pipelines which is essential for metaprogramming optimization.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords Polyhedral Framework, Scalar Dependence, LLVM, Polly

1 Introduction

Advanced high-level loop optimizations, often using polyhedral program models [31], have been shown to effectively improve data locality for image processing [22], iterated stencils [4], Lattice-Boltzmann computations [23], sparse-matrices [30] and even provide some of the leading automatic GPU code generation approaches [5, 32].

While polyhedral optimizers have been shown to be effective in research, but are rarely used in production, at least partly because they are disabled in the most common optimization levels. because none of the established frameworks is enabled by default in a standard optimization setting (e.g. -O3). With GCC/graphite [26], IBM XL-C [8], and LLVM/Polly [15] there are three polyhedral optimizers that work directly in production C/C++/FORTRAN compilers and are shipped to millions of users. While all provide good performance on various benchmarks, one of the last roadblocks that prevents enabling polyhedral optimizers by default is that they generally run as pre-processing pass before the standard -O3 pass pipeline.

Most compilers require canonicalization passes for their detection of single-entry-single-exit regions, loops and iteration count to work properly. For instance, natural loops need recognizable headers, pre-headers, backedges and latches. Since a polyhedral optimizer also makes use of these structures, running the compiler's canonicalization passes is necessary before the polyhedral analyzer. This introduces changes in the intermediate representation that appear as runtime performance noise when enabling polyhedral optimization. At the same time, applying loop optimizations – which often increase code size – early means that no inlining has yet been performed and consequently only smaller loop regions are exposed or future inlining opportunities are prevented.

Therefore, the better position for a polyhedral optimizer is later in the pass pipeline, after all inlining and canonicalization has taken place. At this position the IR can be analyzed without having to modify it and heuristic inlining decisions are not influenced. However, when running late in the pass pipeline, various earlier passes introduce IR changes that polyhedral optimizers do not handle well.

For instance, Listing 1 works entirely on arrays and is easier to optimize than Listings 2 to 4. This is because of the temporaries a and c, which baked-in the assumption that

```
    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. Running 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. Sum-of-products after hoisting the load of A[i].

```
    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. Sum-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].

the *i*-loop is the outer loop. Although the data-flow dependencies are the same, new *false* dependencies are introduced that prevent the scalars a or c from being 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

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

We call dependencies that are induced by a single memory location such as a or c, *scalar dependencies*. Such false dependencies effectively serialize the code into the order it is written, inhibiting or at least complicating most advanced loop optimizations including tiling, strip-mining, and parallelization. Certain specific optimizations can cope 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 [9] do not take privatization opportunities into account. These are fundamentally split into an analysis and a scheduling part. Analysis extracts dependences to determine which execution orders are possible, and the scheduler uses this information to find a new ordering that does not violate any dependence. The analysis result must be correct even for transformations such as loop distribution and -reversal which are not enabled by these transformation-specific techniques. Hence, to allow maximal scheduling freedom, it is desirable to model programs with a minimal number of scalar dependencies. As the semantics of Listings 1 to 4 are identical, a 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 Value Coalescing (Section 2): Map promoted scalars to array elements that are unused to handle cases such as Listings 3 and 4.
- Operand Tree Forwarding (Section 3): Move instruction and memory reads to the statements where they are used to handle cases such as Listing 2.
- An implementation (Section 4) of these techniques in LLVM/Polly.
- An evaluation (Section 5) showing that polyhedral loop optimizations still can be effective in the presence of scalar dependencies.

Sources of Scalar Dependencies   Scalar memory dependencies arise for several reasons and not necessarily introduced by compiler passes.

Single Static Assignment (SSA) -based intermediate representations used in modern compilers cause memory 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.

Loop-Invariant Code Motion (LICM) moves instructions that evaluate to the same value in every loop iteration before the loop, and instructions whose result is only used after the loop, after the loop. The transformation of Listing 1 to Listing 2 is an example for this, but also the temporary promotion of memory location to a scalar as in Listing 3 or 4.
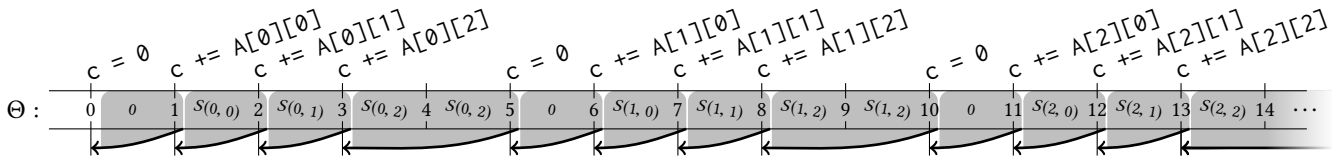
Figure 1. Reaching definition and known content of c.
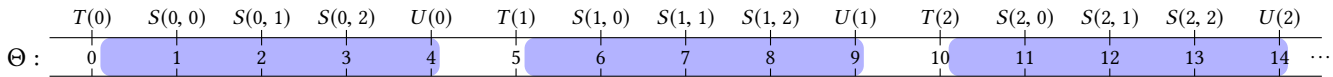
Figure 2. Lifetime of variable c.
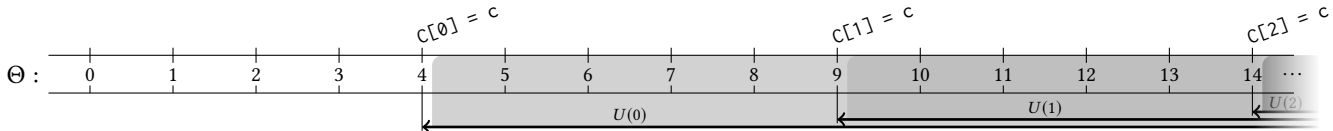
Figure 3. Reaching definition and known contents of C.
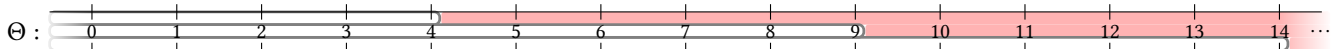
Figure 4. Lifetimes and unused zone of array elements C[0], C[1] and C[2].

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, as uBLAS does [17]. 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.

Manual source optimizations applied by users to not rely on the compiler to do the optimization. For instance, possible aliasing between array C and A in Listing 1 may prevent LICM to be performed by the compiler. Even C++17 does not yet provide annotations (e.g., **restrict**) to declare the absence of aliasing, so programmers may just write Listing 3, or just consider it to be the more natural formulation.

Code generators which are not meant to generate output read by humans, might be inclined to generate pre-optimized code. TensorFlow XLA, for instance, passes code to LLVM with pre-promoted variables in its matrix multiplication kernels [1].

2 Value Coalescing

In this section we present the idea of reusing an array element to store a scalar. We first present the idea on an example (next section) and the general algorithm in Section 2.2.

2.1 Value Coalescing by Example

To illustrate how the coalescing algorithm will work, we apply it on Listing 3 with $N = K = 3$. Figures 1 to 6 visualize

the execution of each statement. Assuming that the execution of each statement takes one time unit, each statement instance has a *timepoint* that marks the time passed since the execution of the first statement. Let a *zone* be a set of one or more intervals between (integer) timepoints. For instance, the zone $(3, 5]$ is the duration between timepoint 3 and 5, excluding 3 itself.

The *reaching definition* of a memory location is the last write to a variable at some timepoint. Figure 1 shows the reaching definition of the variable c. The reaching definition in the zone $]3, 5]$ is $S(0, 2)$ because $S(0, 2)$ writes the value that we would read from c in that zone. If, like it is the case for $S(0, 2)$, a statement reads from and writes to the same location, we assume that the read is effective in the preceding reaching definition. For this reason the definition zones are always left-open and right-closed. Zones can also be unbounded, like the reaching definition zone of the last write $S(2, 2)$ in the example.

Using these reaching definitions, we also know which value a location contains in the definition zone. In case of $T(0)$, this is the constant 0. For cases where there is no direct representation, we use the statement instance itself to represent the value it writes. Timepoints before the first write use the value the variable had before timepoint zero.

The same rules also applies to the array C as shown in Figure 3. The difference is that each array element has its own reaching definition. In this example, each element is written exactly once.

The zone from the definition to its last use is the definition's *lifetime*. If there is no use, then the lifetime is empty. A variable's lifetime is the union of all its definition's lifetimes. Figure 2 shows the lifetime of c. The lifetime for the
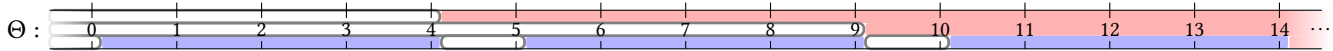
Figure 5. Coalesced memory of `c` and `C[2]`.
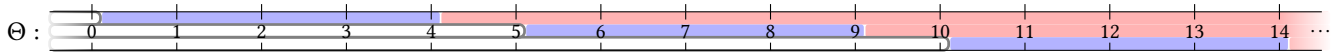
Figure 6. Coalesced memory of `c` into separate elements of `C`.

```
   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];
U:    C[i] = C[i];
   }
```

Listing 5. Sum-of-products (Listing 3) after mapping of the scalar `c` to the array elements `C[i]`.

array C is shown in Figure 4, separately for each of the three elements.

A location's zone is *unused* if there is no lifetime using it. Unused zone intervals usually extend from the last read of a reaching definition to the write of the next definition zone.

It is easy to see that the lifetime of `c` fits into the unused zone of array C. Hence, there is no need to reserve memory for both of them; we can save memory by coalescing both variable's memory to the same space.

One possibility is to store all of `c` into the memory of `C[2]`, as shown in Figure 5. The alternative is to map each lifetime to a different array element, shown in Figure 6. The latter results in the code shown in Listing 5.

For our purpose, this latter variant is preferable. The first reason is that when $U(i)$ reads `c` to store it to `C[i]`, it has to load c's value from `C[i]`. That is, statement $U(i)$ is not required anymore. Secondly, and more importantly, the per-definition lifetimes of `c` are stored at different locations such that there are no false dependencies between them.

The observation that statements can be removed if two adjacent lifetimes use the same storage motivates a simple heuristic: at a statement that starts a lifetime (that is, an array element is written), find another lifetime that ends which can be mapped to the same element. A lifetime can be mapped if the array element's unused zone is large enough for the entire lifetime.

The new lifetime of `C[i]` is the union of the lifetimes of the scalar and the previous lifetime. If there were other unmapped scalars in the program, we could now proceed to map these scalars as well. If two lifetimes overlap it might still be possible to coalesce them. In Figure 2 we annotated the known content at each timepoint. If the lifetimes overlap, but are known to contain the same value, they can still be mapped to the same memory location. This is important for cases such as Listing 4. Here, the writeback of `c` to `C[i]` in

```
1   Knowledge = computeKnowledge();
2   for (w : Writes) {
3     if (!eligibleForMapping(w))
4       continue;
5     t = getAccessRelation(w);
6     Workqueue = getInputScalars(w.Parent);
7     while (!Workqueue.empty()) {
8       s = Workqueue.pop();
9       if (!eligibleForBeingMapped(s, t))
10        continue
11      Candidate = computeKnowledge(s);
12      Proposal = mapKnowledge(Candidate, t);
13      if (isConflicting(Knowledge, Proposal))
14        continue;
15      WriteStmts = { w.Parent | w ∈ WritesTo(s) };
16      for (a :  ReadsTo(s) ∪ WritesTo(s))
17        a.setAccessRelation(Target);
18      Knowledge = Knowledge ∪ Proposal;
19      for (w : WriteStmts)
20        Workqueue = Workqueue ∪ getInputScalars(w);
21    }
22  }
```

Listing 6. Greedy scalar mapping algorithm.

every instance of statement $S(i, j)$ does not end its lifetime, but overlaps with the lifetime of `C[i]`.

2.2 The Greedy Scalar Mapping Algorithm

We have seen the outline of how to coalesce scalars and array elements to avoid dependencies. In this section, we describe a general algorithm that selects the memory to be coalesced. The algorithm's pseudocode is shown in Listing 6. 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 loop nest, 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 a location.
  - The reaching definition derived from the writes.
  - The written value is the known content for the reaching definition's zone.
- All reads from a location.
  - The lifetime zones derived from the reads and the reaching definitions.

- All memory location's unused zones, which is the complement of all lifetimes occupying that location. Alternatively to using the complement, one can compute the zones from each reaching definition's last read to the next write.

All of these can be represented as polyhedra with dimensions tagged as belonging to an array element, timepoint, statement or value.

Every write potentially terminates an unused zone right before it. Some mapping target locations 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 Figure 6). 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 Listing 5). These candidates are used to initialize a queue, with the best candidate coming out first. Prioritized candidates are those that lead to a load from the 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 include being larger than the target location, being defined before the modeled code 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), they must meet the following conditions for every memory location:

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

```
for (int i = ...) {        for (int i = ...) {
    c = ...                    A[i] = ...
T:  use(c);             T:    use(A[i]);
U:  A[i] = g(c);        U:    A[i] = g(A[i]);
}                          }
```

Listing 7. Example when coalescing restricts scheduling possibilities. On the left statements T and U can be executed in exchanged order; after mapping c to A[i] (right) this is not possible anymore.

Note that a write to A or B does not induce a lifetime if that value is never read. Still, such a write to, say A, would overwrite the content of B if coalesced together. Therefore both directions need to be checked.

If there is no conflict, then the proposal is applied. All uses of s are changed to uses of t (line 17). The proposal is used to update the knowledge (line 18) by uniting the lifetimes and known content. Since writes to s have become writes to t, a new unused zone ends at each write, to which variables can be mapped to. As with the original write (line 6), we add possible mapping candidates to the work queue (line 20). Because after mapping there are no more writes to the location of s, we use the writing statements from before the transformation (line 15).

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). As Listing 7 shows, coalescing can also add additional false dependencies. However, loop-carried dependencies block more optimizations that loop-internal ones.

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 8. Thereafter the variable a becomes dead code.

Operand forwarding is the replication of instruction operands defined by operations in other statements. This is possible when the operation's arguments are still available, which can in turn be made available by forwarding them as well such that transitively we get an operand tree.

Besides *pure* instructions whose result only depend on their arguments, replicating a load requires that the location loaded from contains the same value. 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 that the reload does not need to happen from the same location as

```
   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 8. Operand tree forwarding applied on sum-of-products (Listing 2).

```
1   bool canForwardTree(Target, s, Knowledge) {
2     e = Knowledge.findSameContent(s, Target);
3     if (e) return true;
4     if (s is not speculatable)
5       return false;
6     for (op : operands(s))
7       if (!canForwardTree(Target, op, Knowledge))
8         return false;
9     return true;
10  }
```

Listing 9. Determine whether an operand tree is forwardable.

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

Listing 10. Replicate an operand tree to `TargetStmt`.

the original load, and the reloaded value does not even need to be a result of a load.

Listing 9 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 10 can be called to carry-out the forwarding.

Both procedures are recursive functions on the operand tree's children. They are correct on DAGs as well, but lead to duplicated subtrees. `doForwardTree` repeats the traversal of `canForwardTree`, but can assume that every operation can be forwarded. A tree that contains a non-forwardable subtree would require the introduction of a scalar dependency for its value, hence such trees are not forwarded at all.

The method `findSameContent` (line 2) 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 reloaded from these locations. The new load is added to the list of operations executed by the target statement (line 4 of `doForwardTree`).

If the operation is speculatable (line 4 in `canForwardTree`), i.e., its result depends only on its arguments, we also need to check whether its operands are forwardable (line 7) 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 7) and a recursive call on its operands ensures that the operands values are as well available in the target statement (line 9).

4 Implementation

Both transformations, value mapping and operand tree forwarding, have been implemented in *Polly* [15]. Polly inserts itself into the LLVM pass pipeline depending on the `-polly-position` option, as shown in Figure 7, which in the following we shorten to *early* and *late*.

LLVM's early mid-end is (also) meant to canonicalize the 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 canonicalized 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. Polly depends on LLVM's loop detection and its ScalarEvolution analysis, such that a selected set of passes are added before Polly.

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 accesses and statements that have become unnecessary. The vectorizer expects canonicalized IR, so the mid-end passes are run again if Polly generated any code in that 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.

IR-Specific Algorithm Details   Because LLVM-IR is an SSA-based language, a value can be uniquely identified by the operation and the statement instance it is computed in. We use this for known content analysis. For instance, the value a produced by statement $T(i)$ in Listing 2 is represented by $[T(i) \rightarrow a]$. 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 $[T(i) \rightarrow a]$.

In addition to the SSA-value for result of the φ-node, Polly models a φ with an additional memory location as shown in
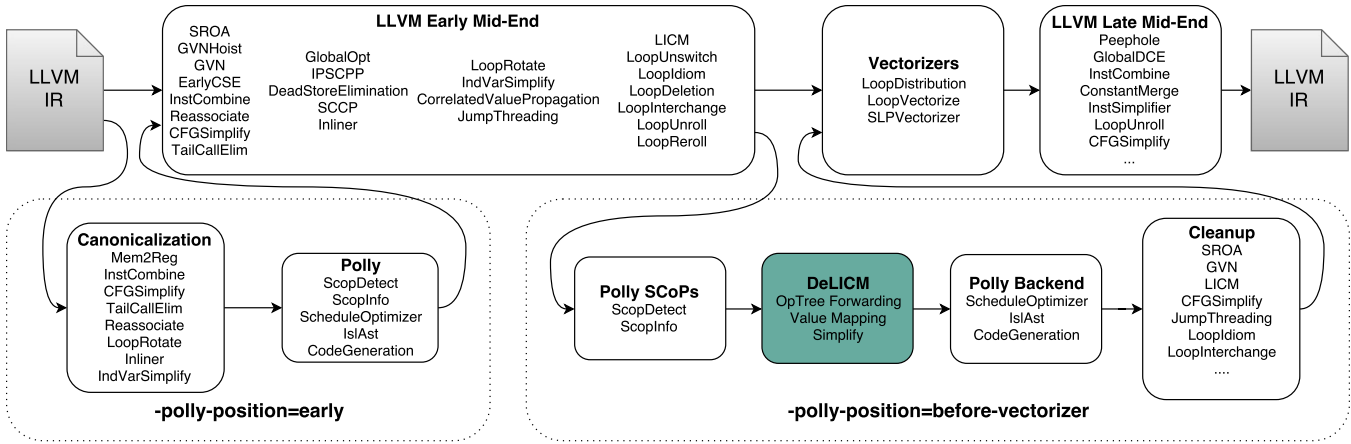
Figure 7. Position of the Polly polyhedral optimizer in the LLVM pass pipeline.

```
pred1:
  br label %basicblock ; write 0.0 to %v.phiops

pred2:
  br label %basicblock ; write 1.0 to %v.phiops

basic_block:
  ; read from %v.phiops
  %v = phi float [0.0, %pred1], [1.0, %pred2]
```

Listing 11. Handling of φ-nodes is analogous to SSA-values, but with multiple writes and a single read to/from a virtual `%v.phiops` variable.

Listing 11. Listing 6 does not assume a single write per scalar, hence is also applicable. The fact that there is only one read from the additional locations helps computing the lifetime. The φ's known content can be replaced by the incoming value in simple cases.

5 Evaluation

The evaluation consists of two parts. In Section 5.1 we analyze the transformations applied by DeLICM itself and which polyhedral optimizations are enabled by it. The measurements are done statically by the compiler without executing the code. The second part in Section 5.2 evaluates the speed-up gained through the enabled transformations.

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 physical and 16 logical cores, but we only evaluate single-thread performance. The hardware details are mostly relevant for the performance tests, but some mid-end 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 317808) using the optimization level `-O3 -march=native -ffast-math` for all tests. All evaluation is done on Polybench/C 4.2.1 beta [24].

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. This allows us to compare the DeLICM result to a reference source without scalar dependences.

Unfortunately, we hit a problem where the combination of the LLVM passes LoopRotate and JumpThreading causes ScalarEvolution to being unable to analyze a loop when an inner loop has no iterations. We modified the Polybench source to avoid this problem (by peeling off the first outer loop), because this is a shortcoming of LLVM's canonicalization, unrelated to Polly. We are in contact with the LLVM community to resolve this in the future.

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 more code movement. For all tests, we use Polybench's `-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 [17]: 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.

Table 1. Number of applied DeLICM transformations at early and late positions in the LLVM pass pipeline. "Forwards" is the number of copied instructions (line 7 of Listing 10), plus the number of reloads from arrays (line 4). "Mappings" is the number of SSA values (plus φ nodes) mapped to array locations (number of scalars that pass the conflict test at line 13 of Listing 6).

Benchmark	early Forwards	early Mappings	late Forwards	late Mappings
Data Mining				
correlation	0	0	1+1	3+3
covariance	0	0	2+0	2+2
Linear Algebra				
gemm	0	0	1+1	0
gemver	0	0	0+2	2+2
gesummv	0	0	0	2+2
symm	0	0	1+4	0
syr2k	0	0	0+2	0
syrk	0	0	1+1	0
trmm	0	0	0+1	0
2mm	0	0	0	2+2
3mm	0	0	0	3+3
atax	0	0	0	1+1
bicg	0	0	0+1	1+1
doitgen	0	0	0	0
mvt	0	0	0	2+2
Solvers				
cholesky	0	0	0	0
durbin	3+3	2+1	0	0
gramschmidt	0	0+3	0	2+2
lu	0	0	0+1	0
ludcmp	0+3	0+11	0+3	1+7
trisolv	0	0	1+1	0
Dynamic Programming				
deriche	8+0	0+4	4+0	0+4
nussinov	-	-	-	-
Shortest Path				
floyd-warshall	0	0	0	0
Stencils				
adi	74+0	0	0+4	0+2
fdtd-2d	0	0	0+3	0
heat-3d	0	0	0+2	0
jacobi-1d	0	0	0+4	0
jacobi-2d	0	0	0+2	0
seidel-2d	0	0	0+6	0
uBLAS C++				
correlation	-	-	1+3	3+0
covariance	-	-	2+2	2+0
gemm	-	-	0+1	1+1
2mm	-	-	0+1	2+2

Table 2. Number of scalar writes in loops before and after DeLICM. Each cell shows the number of SSA-scalar plus the number of φ write accesses. "Post-Opts" are additional post-scheduling optimizations that can be applied with DeLICM compared to without it. The first is the number of additional tiled loops and the number in parentheses the number of additional optimized matrix-multiplications.

| Benchmark	early before	early DeLICM	early Post-Opts	late before	late DeLICM	late Post-Opts
Data Mining						
correlation	0	0	0	4+6	0	+5
covariance	1+0	1+0	0	3+4	1+0	+3
Linear Algebra						
gemm	0	0	0	1+0	0	0 (+1)
gemver	0	0	0	4+4	0	+3
gesummv	0	0	0	2+4	0	+1
symm	0+5	0+5	0	5+2	1+2	0
syr2k	0	0	0	2+0	0	+1
syrk	0	0	0	1+0	0	+1
trmm	0	0	0	1+2	0	+2
2mm	0	0	0	2+4	0	+2 (+2)
3mm	0	0	0	3+6	0	+3 (+3)
atax	0	0	0	1+2	0	+2
bicg	0	0	0	2+2	0	0
doitgen	0	0	0	1+2	1+2	0
mvt	0	0	0	2+4	0	+2
Solvers						
cholesky	0	0	0	0	0	0
durbin	4+5	1+3	0	5+4	5+4	0
gramschmidt	1+5	1+0	0	3+4	1+0	+3
lu	0	0	0	0+2	0	+1
ludcmp	3+18	0	0	4+14	0	0
trisolv	0	0	0	1+4	0+4	0
Dynamic Programming						
deriche	0+28	0+20	0	0+28	0+20	0
nussinov	-	-	-	-	-	-
Shortest Path						
floyd-warshall	0	0	0	0	0	0
Stencils						
adi	0	0	0	0+12	0	+4
fdtd-2d	0	0	0	1+4	0	0
heat-3d	0	0	0	0+4	0	+1
jacobi-1d	0	0	0	0+8	0	+1
jacobi-2d	0	0	0	0+4	0	+1
seidel-2d	0	0	0	0+12	0	0
uBLAS C++						
correlation	-	-	0	3+6	0	+4
covariance	-	-	0	2+4	0	+4
gemm	-	-	0	2+2	0	+2 (+1)
2mm	-	-	0	3+4	0	+4 (+2)
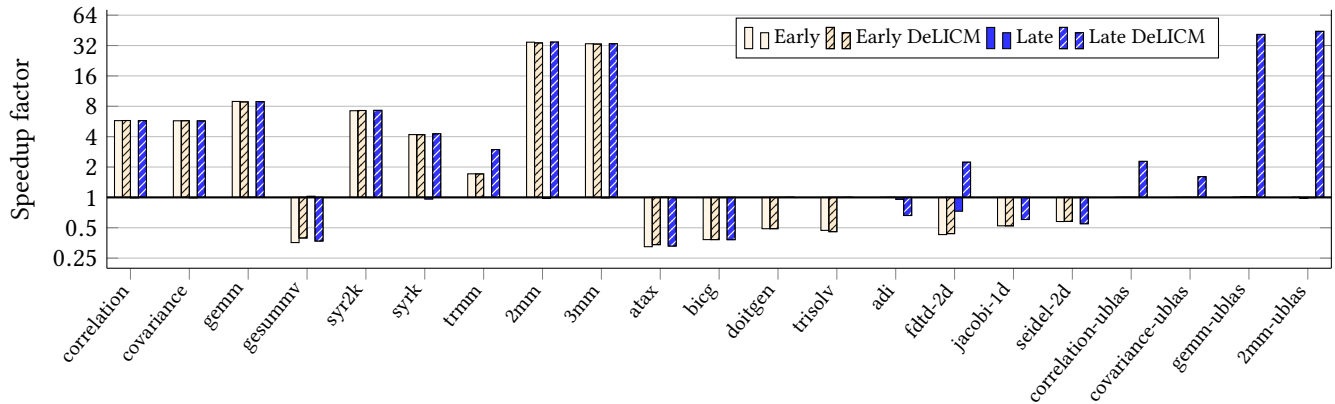
Figure 8. Speedups relative to clang without Polly (-O3) using median execution times. Each experiment ran 5 times, with all median absolute deviations being less than %2 of the median. DeLICM is able to recover the original performance at late position in most cases.

5.1 Static Evaluation

Table 1 shows how many transformations the DeLICM algorithms applies on the benchmarks. To show how effective they were, Table 2 shows the number of scalar writes in loops. We use this measure as a proxy for scalar dependencies. Not every scalar dependency inhibits loops transformations (for instance, using a constant defined before the loop), but every scalar write in a loop causes a false dependency itself in the next iteration, and every dependency involves a write access.

When Polly is at the early position, only few scalar dependencies are present and consequently DeLICM does mostly nothing. At the late position, DeLICM successfully removed all scalar dependencies in 24 (of 31 that had at least one) programs.

One of the trickier cases is atax in which the scalar temporary, called tmp, was manually expanded into an array. The LLVM passes almost entirely recovered the scalar nature of the temporary so it is not obvious to find a mapping location. The GVN partial redundancy elimination introduces φ nodes into stencil iterations to reuse an already loaded value in the next stencil. This effectively serializes the inner loop. To make it parallel again, the φ-nodes have either to be forwarded or traversed to find the original location. Our implementation does the latter.

In the following, we discuss cases where not all scalar dependencies could be removed and explore the reasons for it. The programs symm, durbin and deriche use more scalars in their loops than there are unused array elements. DeLICM does not help in these cases. In covariance, the dependence is caused by a division that cannot be forwarded because LLVM cannot prove that the divisor is non-zero. The LLVM framework provides no property for instructions that can be executed multiple times, if they are executed at least once. We had to fall back to the stricter *speculatable* property which also allows instructions to be executed if they were

not executed at all in the original program. Anyway, Polly is able to extract the performance-critical code and optimize it. The remaining scalar write in gramschmidt is due to a call to the sqrt function whose result is used in an inner loop. LLVM does not consider sqrt speculatable such that the operand tree forwarder does not copy a call to it. LLVM's LoopIdiom pass introduces a memset memory intrinsic into doitgen's code. Although Polly handles memsets, it causes each array element to be split into byte-sized subelements. Our implementation of content analysis does not yet support mixed-size array elements. Similarly, trisolv has some confusion about an array's type. Uses of the same array as 64-bit integer and double-precision floating-point occur in the same function .

Polly failed to recover the array subscripts of an access in nussinov (at early and as well as at late position) and because of it, gives up optimizing it. Polly's access analysis is not invoked. LLVM does not move code out of loops in floyd-warshall, hence DeLICM has nothing to do. The function calls to the overloaded operators in the C++ codes also causes Polly to bail out. After inlining, they are similar to the original Polybench implementation.

5.2 Performance Evaluation

The runtime speed-ups are shown in Figure 8 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. A run of all benchmarks takes in the range of 1 hour to complete.

With Polly executed early in the pipeline ("Early" bars), we gain the highest speed-ups with the matrix-multiplication family (gemm, 2mm, 3mm) due to a special optimization path for it. The benchmarks using uBLAS cannot be optimized because they contain not-yet inlined function calls. The benchmarks gemver, symm, mvt, cholesky, durbin, gramschmidt,

lu, ludcmp, deriche, floyd-warshall, nussinov, heat-3d and jacobi-2d are not shown in the figure because neither the pipeline position nor DeLICM have a significant effect on their execution speeds, in other cases Polly's transformations even cause slowdowns. Our contribution is to enable transformations, not about which one to select. Therefore any change in performance relative to Polly switched off is a sign that some transformation has been performed and therefore is a success for DeLICM. We continue working on the heuristics that improve performance, or at least do not regress performance.

Executing Polly at the late position causes the performance to fall back to the performance without Polly ("Late"). Generally, this is because Polly does not generate any code because of new scalar dependencies. Enabling DeLICM ("Late DeLICM") almost always restores the original performance. For some benchmarks, we even got a (higher) speed-up where there was none before: trmm and fdtd-2d. Thanks to the inlining having happened at the late position, the C++ benchmarks using uBLAS can now be optimized as well. 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. The only benchmark where the performance could not be recovered are doitgen and trisolv, caused by scalar dependencies that could not be removed.

Applying DeLICM at the beginning of the LLVM pipeline ("Early DeLICM"), without LICM and GVN, has little effect as the Polybench benchmarks avoid scalar dependencies in the source.

6   Related Work

We can categorize most work on avoiding false dependencies into two categories: memory duplication and ignoring non-flow dependencies.

The first category provides additional memory such that there are fewer false dependencies between unrelated computations. Techniques such as renaming, scalar expansion and node splitting [2, 18] have been proposed even before the appearance of the polyhedral model. Taken to the extreme, every statement instance writes to its own dedicated memory [14]. Like in SSA, every element is only written once and therefore this mode is also called *Dynamic Single Assignment* (DSA) [28] or *Array SSA* [16]. This eliminates all false dependencies at the cost of potentially greatly increasing memory usage. Refinements of this technique only duplicate a private copy per thread [20]. This can solve the problem for parallelism but does not enable inner-thread scheduling such as tiling. [27, 29] propose privatization on demand, when an optimization finds restricting dependencies.

There is the possibility to contract array space after scheduling [6, 7, 11, 19, 25, 34]. However, the re-scheduled program

may inherently require more memory than the source program, especially if the scheduler is unaware that its decisions 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* [21] allows loop-fusion. Tiling and parallelism can be recovered using *live-range reordering* [3, 33].

Both come with the disadvantage that they are only enable specific optimizations. 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.

A conflict set as presented in [7, 11] can be useful to determine whether a scalar is conflicting with an array (Listing 6 line 13). For the purpose of DeLICM we additionally need to analyze the stored content. One participant of the conflict set is always a scalar such that a full-array conflict set is not needed.

7   Conclusion

This paper presented 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 two 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 scalar to unused memory locations. This problem is akin to register allocation, with mapping targets being array elements instead of processor registers – an NP-complete problem [10]. 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 where their result is needed or reloads values from array elements that are known to contain the same value.

We implemented both algorithms in Polly, a polyhedral optimizer for the LLVM intermediate representation. The experiments have shown that DeLICM 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 with templates.

A Artifact Appendix

A.1 Abstract

The artifact consists of a Github repository where each branch represents an experiment. The branches contain the Polly source code (unmodified, since all required changes have been upstreamed), and a script that compiles LLVM with Polly and runs the experiments.

In addition, we provide the experimental results used for evaluation (Section 5) and a script to extract and compare results.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** DeLICM - avoid scalar dependencies in the polyhedral model.
- **Program:** Polybench/C 4.2.1 beta [24]. However, we use a slightly modified version available at github.com/Meinersbur/test-suite/tree/cgo18/Performance.
- **Compilation:** Any C++11-compatible compiler to boostrap Clang which then compiles the benchmarks.
- **Binary:** Benchmark binaries included in result branches for our platform (Intel Haswell), but we strongly recommend recompiling for each individual platform.
- **Data set:** EXTRALARGE_DATASET (predefined by Polybench).
- **Run-time environment:** Any Unix system supported by LLVM.
- **Hardware:** Any platform supported by LLVM. For comparison with our reference results, using a similar platform as described in Section 5 is advised (Intel Xeon E5-2667 v3).
- **Output:** A JSON file which can be analyzed by an additional script. The script prints the median/average/min/max of the measurements and the relative difference between experiments to the console. Commands that reproduce the data in the article are provided.
- **Experiments:** Python scripts that bootstrap clang, compile and run the benchmarks.
- **Publicly available?:** Yes, in Github repository.

A.3 Description

A.3.1 How Delivered

The artifact is available for download from https://github.com/Meinersbur/cgo18. Cloning the repository takes around 30MB of disk space.

```
$ git clone https://github.com/Meinersbur/cgo18
```

A.3.2 Hardware Dependencies

Any platform supported by LLVM, see llvm.org/docs/GettingStarted.html#hardware.

The precompiled experiments were compiled for/on Red Hat Enterprise Linux Server 6.7 x86_64 (Intel Haswell) and may or may not work on different platforms. For optimal results, we recommend recompiling for each platform.

A.3.3 Software Dependencies

All requirements needed to compile LLVM, see llvm.org/docs/GettingStarted.html#software. In addition, the

Table 3. Description of all predefined experiments.

Branch	Position	DeLICM	Description
A10_nopolly	-	-	without Polly
A20_polly	late	enabled	current default
A30_polly_early	early	disabled	
A40_polly_late	late	disabled	
A50_polly_early_delicm	early	enabled	

provided scripts assume an environment where the following software is installed:

- CMake 3.6.3 or later (cmake.org)
- Ninja (ninja-build.org)
- Python 2.7 and 3.4 or later (python.org)
- pandas (pandas.pydata.org)
- Git 2.5 or later (git-scm.com)
- OpenSSH client (openssh.com)

A.4 Installation

No installation required.

A.5 Experiment Workflow

The script cgo.py provided in the master branch of the cgo18 repository performs the check-out of LLVM, compilation, benchmark execution and evaluation of the measurements. To run, execute:

```
$ cd cgo18
$ python3 ./cgo.py 2>&1 | tee log.txt
```

This process may take around 21 hours to execute and consumes about 7 GB of disk space. The script prints results in-between other output, so the log.txt might be helpful.

A.6 Evaluation and Expected Result

The cgo.py script will print the following information to the console:

- Comparison of the obtained results to our results ("leone").
- Number of transformations as in Table 1.
- Number of scalar dependencies and post-scheduling transformations as in Table 2.
- Speed-ups as in Figure 8.

A.7 Experiment Customization

The cgo.py script calls two other scripts to compile and run the benchmarks (gittool.py) and extract the results (execcmp.py). These commands can also be invoked manually.

Each experiment is described by a branch in the repository with a line starting with Exec: describing the experiment to execute. The predefined experiments are shown in Table 3.

An experiment can be modified by changing the Exec: in the commit message of the branches latest commit. We recommend creating a new branch for each new experiment. The available options can be seen using:

```
$ python3 ./gittool.py runtest --help
```

The source code of Polly, and the script being executed (run-test-suite.py) are as well part of the commit, such that these can be modified as well., e.g. to test modifications to the compiler that are not exposed via a command-line switch.

Table 4. Hashes (first 6 digits) to the commits describing an experiment and the hashes resulting from the execution on Monte Leone.

Branch	Experiment	"leone"
A10_nopolly	88faaff	98fafc2
A20_polly	905f739	895c01f
A30_polly_early	9274722	a7d8e2d
A40_polly_late	27c0354	3cd76a7
A50_polly_early_delicm	fbf19b5	1830678
master		87eb3a

The experiment of the currently checked-out branch is executed using:

```
$ python3 ./gittool.py execjob
```

This will execute the command in the Exec: line, then commit the result to a new branch that is prefixed with the host machine's hostname.

To extract measurements, the script execcmp.py can read the results from a result branch. For instance, the command

```
$ ./execcmp.py leone_A10_nopolly
```

prints the execution time of each benchmark of experiment "A10_nopolly" executed on the machine "leone" (The machine used for our evaluations, http://www.cscs.ch/computers/monte_leone/index.html). The command

```
$ ./execcmp.py leone_A10_nopolly vs leone_A20_polly
```

compares the execution times of the benchmarks with and woutput Polly.

More details about the experiment execution system can be found in the README.md in the master branch of the cgo18 repository. The system also includes a command to create new experiments from scratch and a buildbot component that queues and executes jobs but are not necessary to reproduce the results.

A.8   Notes

The git sha1 hashes allow to uniquely identify each experiment and benchmark result. The data for this paper were taken from experiments with the hashes shown in Table 4.

Acknowledgments

References

[1] Annanay Agarwal. 2017. Enable Polyhedral Optimizations in XLA through LLVM/Polly. Google Summer of Code 2017 final report. (2017). http://pollylabs.org/2017/08/29/GSoC-final-reports.html

[2] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *Transactions on Programming Languages and Systems (TOPLAS)* 9, 4 (Oct. 1987), 491–542. https://doi.org/10.1145/29873.29875

[3] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. 2013. Improved Loop Tiling Based on the Removal of Spurious False Dependences. *Transactions on Architecture and Code Optimization (TACO)* 9, 4, Article 52 (Jan. 2013), 52:1–52:26 pages.

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling Stencil Computations to Maximize Parallelism. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–11.

[5] Muthu Baskaran, Jj Ramanujam, and P Sadayappan. 2010. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*. Springer, 244–263.

[6] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. Automatic Storage Optimization for Arrays. *Transactions on Architecture and Code Optimization (TACO)* 38, 3, Article 11 (April 2016), 11:1–11:23 pages.

[7] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. 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)*. ACM, New York, NY, USA, 526–538.

[8] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, 343–352.

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Notices* 43, 6, 101–113. http://pluto-compiler.sourceforge.net

[10] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation via Coloring. *Computer Languages* 6, 1 (1981), 47–57.

[11] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. 2016. Extended Lattice-Based Memory Allocation. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 218–228.

[12] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem – Part I. One-dimensional Time. *International Journal of Parallel Programming* 21, 6 (Oct. 1992), 313–347.

[13] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem – Part II. Multidimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[14] Paul Feautrier. 2014. Array Expansion. In *International Conference on Supercomputing 25th Anniversary Volume (SC '14)*. ACM, New York, NY, USA, 99–111.

[15] Tobias Grosser, Armin Grösslinger, and Christian Lengauer. 2012. Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012). http://polly.llvm.org

[16] Kathleen Knobe and Vivek Sarkar. 1998. Array SSA Form and Its Use in Parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 107–120. https://doi.org/10.1145/268946.268956

[17] Mathias Koch and Joerg Walter. 2017. Boost uBLAS. (7 Sept. 2017). http://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas/doc/index.html

[18] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/567532.567555

[19] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel Comput.* 24, 3 (1998), 649–671.

[20] Zhiyuan Li. 1992. *Array Privatization: A Loop Transformation for Parallel Execution.* Technical Report 9226. Univ. of Minnesota.

[21] Sanyam Mehta and Pen-Chung Yew. 2016. Variable Liberalization. *Transactions on Architecture and Code Optimization (TACO)* 13, 3, Article 23 (Aug. 2016), 23:1–23:25 pages.

[22] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic Optimization for Image Processing Pipelines. In *SIGPLAN Notices*, Vol. 50. ACM, 429–443.

[23] Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. 2015. An Optimizing Code Generator for a Class of Lattice-Boltzmann Computations. *Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 14.

[24] Louis-Noel Pouchet and Tomofumi Yuki. 2016. Polybench 4.2.1 beta. (2016). Retrieved 2017-07-07 from https://sourceforge.net/projects/polybench

[25] Fabien Quilleré and Sanjay Rajopadhye. 2000. Optimizing Memory Usage in the Polyhedral Model. *Transactions on Architecture and Code Optimization (TACO)* 22, 5 (Sept. 2000), 773–815.

[26] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. Graphite Two Years After: First Lessons learned from Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW '10)*.

[27] Konrad Trifunovic, Albert Cohen, Ladelski Razya, and Feng Li. 2011. Elimination of Memory-Based Dependences for Loop-Nest Optimization and Parallelization. In *3rd Workshop on GCC Research Opportunities (GROW '11)*. Chamonix, France.

[28] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. 2005. Transformation to Dynamic Single Assignment Using a Simple Data Flow Analysis. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*, Kwangkeun Yi (Ed.). Springer, Berlin, Heidelberg, 330–346.

[29] Nicolas Vasilache, Benoit Meister, Albert Hartono, Muthu Baskaran, David Wohlford, and Richard Lethin. 2012. Trading Off Memory For Parallelism Quality. In *International Workshop on Polyhedral Compilation Techniques (IMPACT '12)*.

[30] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and Data Transformations for Sparse Matrix Code. In *SIGPLAN Notices*, Vol. 50. ACM, 521–532.

[31] Sven Verdoolaege. 2016. *Presburger Formulas and Polyhedral Compilation*. Technical Report. https://lirias.kuleuven.be/handle/123456789/523109

[32] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54.

[33] Sven Verdoolaege and Albert Cohen. 2016. Live Range Reordering. In *International Workshop on Polyhedral Compilation Techniques (IMPACT '16)*. Prague, Czech Republic.

[34] Doran K. Wilde and Sanjay Rajopadhye. 1993. *Allocating Memory Arrays for Polyhedra*. Research Report RR-2059. Inria.