Elimination of Memory-Based Dependences for Loop-Nest Optimization and Parallelization: Evaluation of a Revised Violated Dependence Analysis Method on a Three-Address Code Polyhedral Compiler

Konrad Trifunovic[1], Albert Cohen[1], Razya Ladelsky[2], and Feng Li[1]

[1] INRIA Saclay – Île-de-France and LRI, Paris-Sud 11 University, Orsay, France
{albert.cohen, konrad.trifunovic, feng.li}@inria.fr
[2] IBM Haifa Research, Haifa, Israel
razya@il.ibm.com

Abstract

In order to preserve the legality of loop nest transformations and parallelization, data-dependences need to be analyzed. Memory dependences come in two varieties: they are either data-flow dependences or memory-based dependences. While data-flow dependences must be satisfied in order to preserve the correct order of computations, memory-based dependences are induced by the reuse of a single memory location to store multiple values. Memory-based dependences reduce the degrees of freedom for loop transformations and parallelization. While systematic array expansion techniques exist to remove all memory-based dependences, the overhead on memory footprint and the detrimental impact on register-level reuse can be catastrophic. Much care is needed when eliminating memory-based dependences, and this is particularly essential for polyhedral compilation on three-address code representation like the GRAPHITE pass of GCC.

We propose and evaluate a technique allowing a compiler to ignore some memory-based dependences while checking for the legality of a given affine transformation. This technique does not involve any array expansion. When this technique is not sufficient to expose data parallelism, it can be used to compute the minimal set of scalars and arrays that should be privatized. This form of privatization is guaranteed to preserve the per-hardware-thread memory footprint of the original program, unlike systematic array expansion techniques. Our method relies on array dataflow analysis and on an extension of the violated dependence analysis technique implemented in GRAPHITE. It has a minimal impact on compilation time and provides a critical guarantee: while working on three-address code, no affine transformation or parallelization opportunity is lost w.r.t. an implementation of the polyhedral model based on a high-level abstract syntax. This result confirms the interest and potential of low-level polyhedral compilation. The paper details the algorithm, the current state of the implementation, and experimental results showing the benefits on automatic parallelization of the PolyBench suite.

1 Introduction

The problem of deciding which loop nest optimizations to apply is the notoriously difficult optimization problem that a modern compiler has to face [11, 9, 4, 24, 23]. The well known approach to this problem is the polyhedral compilation framework [11] aimed to facilitate the construction and exploration of loop transformation sequences and parallelization strategies by mathematically modelling memory accesses patterns, loop iteration bounds, and instruction schedules.

Each program transformation needs to be safe – the semantics of the original imperative program cannot be changed. In order to preserve legality, data-dependences [1] need to be analyzed. There are two categories of data-dependences: data-flow dependences and memory-based dependences. While preserving data-flow dependences is always mandatory, spurious memory-based dependences can be ignored or removed.

So called *true* or *dataflow* [1] dependences are imposing an ordering constraints between write and read operations – they should always be preserved, since this preserves the right producer-consumer ordering, which in turn guarantees correctness of computations.

Memory-based dependences[3] are induced by the reuse of the same variable to store multiple (temporary) values. Spurious scalar dependences not only increase the total number of dependences that need to be dealt with (having an impact on compilation time), but, most importantly, they reduce the degree of freedom available to express effective loop transformations and parallelization.

Memory-based dependences could be removed by introducing new memory locations, i.e. *expansion* of the data structures [5]. While the *expansion* approaches might remove spurious memory-based dependences, they have to be avoided whenever possible due to their detrimental impact on cache locality and memory footprint.

Polyhedral loop nest optimization and parallelization is traditionally implemented on top of rich, high-level abstract syntax trees. The GRAPHITE pass in GCC is an exception, as it operates on GIMPLE – GCC low-level three-address intermediate code [10]. Designing a polyhedral compilation framework on three-address code exacerbates the problem of spurious memory dependences even further, since the gimplification [4] process introduces many temporary variables.

Providing loop nest optimizations directly on a three-address code provides many benefits. It enables a tight integration of loop optimization techniques with GCC downstream passes, including an automatic vectorization, automatic parallelization and memory optimizations. Efficiently solving the problem of spurious memory dependences enables us to take an advantage of all the GCC passes operating on a three-address low-level code without paying any additional penalty.

In this paper, we show a technique that detects those memory-based dependences that could be ignored when checking for a legality of transformation. If memory-based dependence could not be ignored, it proposes an expansion or privatization that would enable a transformation.

Using proposed technique, we can get rid of many memory-based dependences, either those induced by the lowering of a source program into GIMPLE or simply introduced by a programmer. Only if it cannot be ignored, it is removed through scalar/array expansion. This is excellent news to many loop transformation experts, as it circumvents a well known difficulty with polyhedral compilation techniques.

Proposed technique could be used outside GCC (GRAPHITE) compiler as well. A good target is a source-to-source polyhedral parallelizer PLUTO [4] for example. Our method relies on array dataflow analysis, implemented in ISL by Verdoolaege [22] using

[3] anti and output dependences [1]

[4] *gimplification* is a GCC jargon term denoting the lowering of a high-level AST into a low-level GIMPLE internal representation

parametric integer linear programming, and on an extension of the violated dependence analysis technique which is already implemented in GRAPHITE.

2 State of the art

Memory-based data dependences are known to hamper possible parallelization and loop transformation opportunities. A well known technique for removing spurious data dependences is to *expand* data structures – assigning distinct memory locations to conflicting writes. An extreme case of data expansion is *single-assignment* [8, 7] form, where each memory location is assigned only once.

There is a trade-off between parallelization and memory usage: if we expand maximally, we will get the maximal degree of freedom for parallelization and loop transformations, but with a possibly huge memory footprint. If we choose not to expand at all, we will save memory, but our parallelization or loop transformation possibilities would be limited.

Currently, there are two general approaches to handling memory dependences:

– Perform a maximal expansion, do a transformation, and then do an array contraction which minimizes the memory footprint. Approaches like [14], [13], [6] fall into this category. This approach gives the maximal degree of freedom for parallelization or loop transformation, but an array contraction phase is not always capable of optimizing the memory footprint.
– Control the memory expansion phase by imposing constraints on the scheduling. Approaches like [5], [16] fall into this category. This category of approaches tries to optimize the memory footprint, but it might restrict schedules, thus loosing optimization opportunities.

Our approach takes the lazy-expansion strategy: we do not expand memory before transformation. When checking for transformation legality, we simply ignore all memory based dependences. Only after applying a given transformation, we perform a violation analysis to check which memory based dependences might have been violated, and we propose to expand memory or to change a schedule.

By taking our approach, we are combining the best from two mentioned approaches: we do not perform a full expansion before a transformation and we do not restrict the transformation too early. The lazy-expansion strategy is not used to compute transformations automatically, as it is done in techniques using linear programming approach [9, 4], instead, it is used in an *iterative enumeration* of possible schedules as it is done in [19].

3 Motivating example

Consider a simple numerical kernel – the infamous matrix multiplication – given in a Figure 1. If we are to analyze this code at the source-code level, we will get the dependence graph shown in Figure 7. It contains both dataflow [5], and memory-based [6] dependences. Those data dependences do not prevent loops `'i'` and `'j'` to be permuted.

If we want to compile this source code in GRAPHITE, things are more complicated. After source code is transformed into GIMPLE it goes through many optimization

[5] true, a.k.a. read-after-write
[6] write-after-write and write-after-read

Fig. 1. matrix multiplication

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    {
S1:   A[i][j] = 0;
      for (k = 0; k < N; k++)
S2:     A[i][j] += B[i][k] * C[k][j];
    }
```

Fig. 2. after PRE

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

Fig. 3. Legal execution order

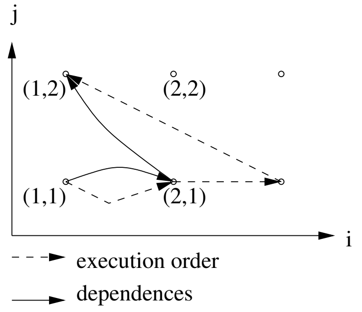
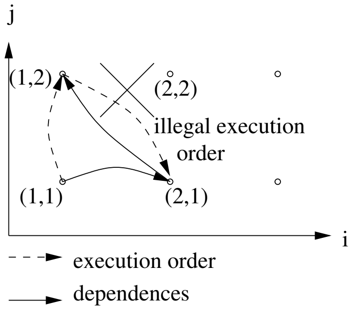


Fig. 4. Illegal execution order



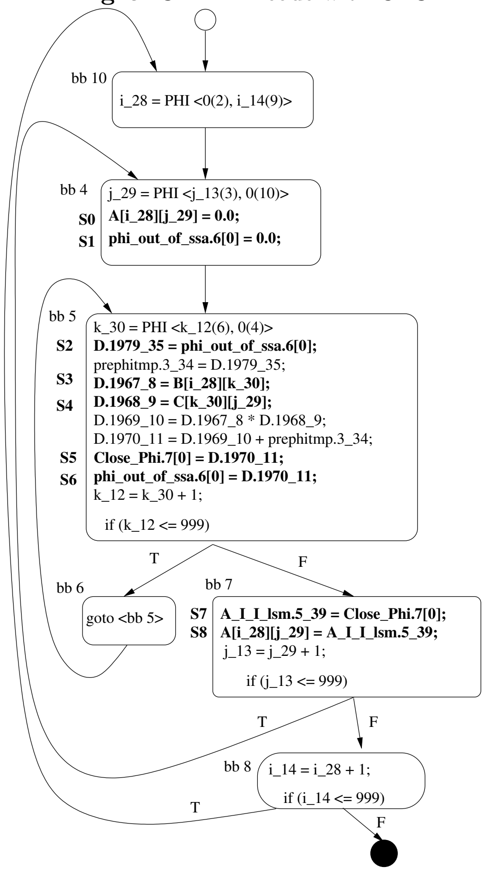
passes until it reaches GRAPHITE. One of those passes is PRE (Partial Redundancy Elimination) which does the following scalar optimization: instead of accumulating a values into an array, it initializes a scalar value, accumulates values into that scalar and then stores the scalar into an array element. Conceptually, the idea is shown in Figure 2.

That is a very good scalar optimization, but it makes things much harder for GRAPHITE to analyze. The code seen by GRAPHITE is shown in Figure 5.

A new dependence graph for the GIMPLE code is shown in Figure 6. Not only has the data dependence graph become more complex, but it is structurally different from the dependence graph seen by a source-to-source compiler. After introducing a scalar into the loop, a new write-after-write dependence on statement S_1 has been introduced: $\delta^{WAW}_{S_1 \to S_1}$. This dependence stems from the fact that the same temporary scalar value is overwritten in each iteration of the containing loop. This dependence has to be respected, which forces sequential execution. Figure 3 shows that if we execute the code in a sequential manner, according to original loop nesting (loop i as outermost, loop j as innermost), then dependences would be preserved. If we try to interchange loops i and j, we would invert a dependence constraint, thus violating the write-after-write dependence on the scalar. This is shown in Figure 4. Currently GRAPHITE would not allow interchanging loops i and j.

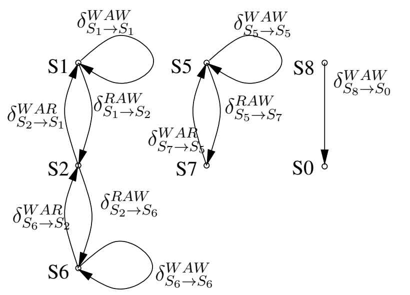
But our intuition tells us that it is legal to interchange loops i and j and still have a correct output code. An essential observation is that some memory based dependences (write-after-write and write-after-read) could be ignored when performing some transformations. But how do we determine when it is safe to ignore some dependences?

Fig. 5. GIMPLE code with CFG



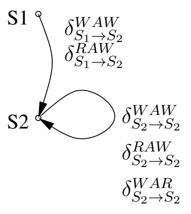
```
bb 10
    i_28 = PHI <0(2), i_14(9)>

bb 4
    j_29 = PHI <j_13(3), 0(10)>
S0  A[i_28][j_29] = 0.0;
S1  phi_out_of_ssa.6[0] = 0.0;

bb 5
    k_30 = PHI <k_12(6), 0(4)>
S2  D.1979_35 = phi_out_of_ssa.6[0];
    prephitmp.3_34 = D.1979_35;
S3  D.1967_8 = B[i_28][k_30];
S4  D.1968_9 = C[k_30][j_29];
    D.1969_10 = D.1967_8 * D.1968_9;
    D.1970_11 = D.1969_10 + prephitmp.3_34;
S5  Close_Phi.7[0] = D.1970_11;
S6  phi_out_of_ssa.6[0] = D.1970_11;
    k_12 = k_30 + 1;

    if (k_12 <= 999)

bb 6
    goto <bb 5>

bb 7
S7  A_I_I_lsm.5_39 = Close_Phi.7[0];
S8  A[i_28][j_29] = A_I_I_lsm.5_39;
    j_13 = j_29 + 1;

    if (j_13 <= 999)

bb 8
    i_14 = i_28 + 1;

    if (i_14 <= 999)
```

Fig. 6. Data Dependence Graph



$\delta^{WAW}_{S_1\to S_1}$
S1
$\delta^{WAR}_{S_2\to S_1}$
$\delta^{RAW}_{S_1\to S_2}$
S2
$\delta^{WAR}_{S_6\to S_2}$
$\delta^{RAW}_{S_2\to S_6}$
S6
$\delta^{WAW}_{S_6\to S_6}$
$\delta^{WAW}_{S_5\to S_5}$
S5
$\delta^{RAW}_{S_5\to S_7}$
$\delta^{WAR}_{S_7\to S_5}$
S7
S8
$\delta^{WAW}_{S_8\to S_0}$
S0

Fig. 7. Matmult Data Dependence Graph



S1
$\delta^{WAW}_{S_1\to S_2}$
$\delta^{RAW}_{S_1\to S_2}$
S2
$\delta^{WAW}_{S_2\to S_2}$
$\delta^{RAW}_{S_2\to S_2}$
$\delta^{WAR}_{S_2\to S_2}$

On the other hand, let us consider an example where an user has written a code in Figure 2 manually and provided it to the compiler. As an example let us consider the legality of loop distribution transformation shown in Figure 8. This transformation would not be legal, and the compiler legality checker would prevent this transformation. If we allow an expansion of the scalar '`t`', as shown in Figure 9, the distribution transformation would be legal.

In the following sections we show how to formally prove which memory-based dependences could be ignored (as in the example of loop interchange) and which could not. In the case where a transformation does not respect memory based dependences (as in the example of loop distribution) a memory expansion could remove this restriction.

We show how an instance-wise variable live range analysis can be used to enable a *lazy* memory expansion scheme – memory is expanded only if absolutely necessary for the correctness of the transformation.

4 Framework

Polyhedral compilation traditionally takes as an input a dependence graph, constructs a *legal transformation* [7] and generates the code.

[7] usually using a mathematical framework based on linear programming

Fig. 8. illegal - loop distribution without an expansion

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

Fig. 9. legal - loop distribution with an expansion

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

Polyhedral compilation based on *violation analysis* [21] does the same, except that it does not necessarily build a legal transformation in one step. Instead, it proposes one transformation and then it checks for data dependence violations that might be induced by this transformation. It tries to correct those (possible) data dependence violations by applying a correction on the transformation. It might fail to correct the transformation, thus selecting a new possible transformation might be necessary. This might lead to an iterative process, as shown in Figure 10.

We build over the aforementioned *violation analysis* and modify it slightly. Instead of taking into an account all dependences, we only check that a given transformation respects *data-flow* dependences. We do not check memory-based dependences for possible violation.

Since a program transformation might change an execution order of instructions, we cannot guarantee that original values written into specific memory location are not destroyed before they could be read by an appropriate instruction. In order to maintain this property, we first define *live ranges* for all values generated in an original program. Then we check whether a transformation would respect those value live ranges. If we prove that all value live ranges are still respected, then the transformation is legal.

If a transformation destroys value live range set for a memory location, it is still possible to correct this: we could choose to replicate (memory expand) those memory locations. This leads to a controllable memory expansion scheme, in which we only expand when there is a need for expansion. A complete scheme of our approach is shown in Figure 10. Some notation and technical details are shown in subsequent subsections.

4.1 Basic notation and concepts

Polyhedral model The scope of the polyhedral program analysis and manipulation is a sequence of loop nests with constant strides and affine bounds. It includes non-perfectly nested loops and conditionals with boolean expressions of affine inequalities [11].

The maximal *Single-Entry Single-Exit* (SESE) region of the *Control Flow Graph* (CFG) that satisfies those constraints is called a *Static Control Part* (SCoP) [11, 4]. GIMPLE statements belonging to the SCoP should not contain calls to functions

Fig. 10. A flow of polyhedral compilation based on violation analysis

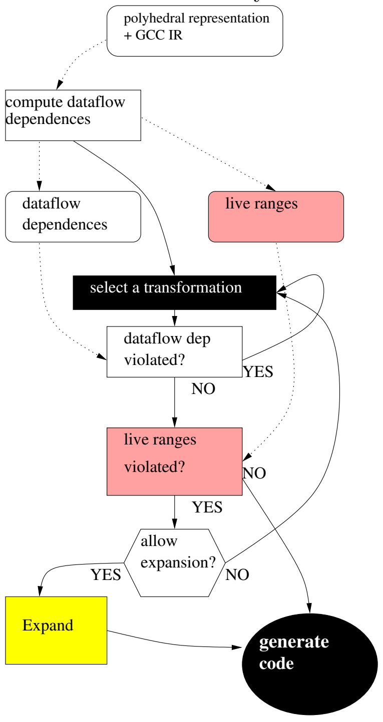
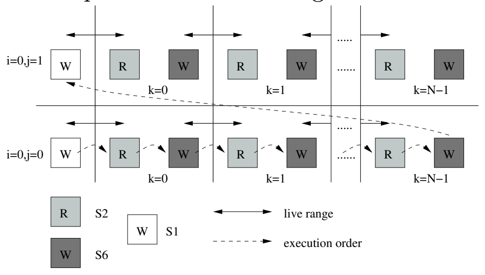


Fig. 11. Write/Read instruction interleaving and respective value live ranges



with side effects (`pure` and `const` function calls are allowed) and the only memory references that are allowed are accesses through arrays with affine subscript functions. SCoP control and data flow are represented with three components of the polyhedral model [11, 4, 15]:

Iteration domains capture dynamic instances of instructions — all possible values of surrounding loop induction variables — through a set of affine inequalities. Each dynamic instance of an instruction S is denoted by a pair $(S, \mathbf{i})$ where $\mathbf{i}$ is the *iteration vector* containing values for the loop induction variables of the surrounding loops, from outermost to innermost. If an instruction S belongs to a SCoP then the set of all iteration vectors $\mathbf{i}$ relevant for S can be represented by a polytope: $\mathcal{D}_S = \big\{\mathbf{i} \mid \mathrm{D}_S \times (\mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\big\}$ which is called the *iteration domain* of S, where $\mathbf{g}$ is the vector of *global parameters* whose dimension is $d_{\mathbf{g}}$. Global parameters are invariants inside the SCoP, but their values are not known at compile time (parameters representing loop bounds for example).

Data references capture memory locations of array data elements on which GIMPLE statements operate. In each SCoP, by definition, the memory accesses are performed through array data references. A scalar variable can be seen as a zero-dimensional array. The data reference polyhedron $\mathcal{F}$ encodes the *access function* mapping iteration vectors in $\mathcal{D}_S$ to the array subscripts represented by the vector $\mathbf{s}$: $\mathcal{F} = \big\{(\mathbf{i}, \mathbf{s}) \mid \mathrm{F} \times (\mathbf{i}, \mathbf{s}, \mathbf{g}, 1)^T \geq \mathbf{0}\big\}$.

Scheduling functions are also called scattering functions inside GRAPHITE following CLooG's [3] terminology. While iteration domains define the set of all dynamic instances of an instruction, they do not describe the execution order of those instances. In order to define the execution order we need to give to each dynamic instance the execution time (date) [9,12]. This is done by constructing a *scattering polyhedron* representing the relation between iteration vectors and time stamp vector $\mathbf{t}$: $\theta = \big\{(\mathbf{i}, \mathbf{t}) \mid \Theta \times (\mathbf{i}, \mathbf{t}, \mathbf{g}, 1)^T \geq \mathbf{0}\big\}$.

Dynamic instances are executed according to the lexicographical ordering of the time-stamp vectors. By changing the scattering function, we can reorder the execution order of dynamic iterations, thus performing powerful *loop transformations*.

A dependence graph $G = (V, E)$ is the graph whose vertices are statements $V = \{S_1, S_2, \ldots, S_n\}$ and whose edges $e \in E$, from S_i to S_j, are representing scheduling constraints between statement instances of S_i and S_j. Those scheduling constraints are caused by data dependences. Dependence edges e are labelled by dependence polyhedra $\delta^{S_i \to S_j}$. Dependence polyhedra describe, in a closed form linear expression, pairs of statement instances whose relative execution order should be preserved: an instance of statement S_i should be executed before an instance of statement S_j [20]. An example of a dependence graph is shown in Figure 6.

Value live ranges An execution trace of a sequential program can be seen as an interleaving of read and write instructions. During an execution of a loop, values are produced and stored into memory locations. Those values are then read by subsequent instructions. The value stored in a memory location is alive until it is destroyed by a subsequent write to the same memory location. We are interested in formally analyzing and modelling value *live ranges* inside loops, using the polyhedral model. Given GIMPLE code in Figure 5 we can model the execution trace and instances of live ranges as shown graphically in Figure 11.

Consider a value v written into a memory cell $M[b]$ by an instruction instance $w = < S_{LW}, \mathbf{i_{LW}} >$. We can compute a set of instruction instances $R = \{< S_R, \mathbf{i_{LR}} >\}$ such that there is a direct data-flow of value v from instance w to instances in set R.

We use the polyhedral representation to represent, in a compact manner, a set of instances of value live ranges for a given memory location. Each instance of a value live range is a tuple, describing a write and read instruction instances.

We consider a set of value live range tuples $L = \{(< S_{LW}, \mathbf{i_{LW}} >, < S_R, \mathbf{i_{LR}} >)\}$. We want to have a closed form expression that summarizes all instances in this set. We can decompose the set L into a set of convex polyhedra, where each polyhedron describes live range instances for a pair of statements: $\lambda^{S_{LW} \to S_R} = \{(\mathbf{i_{LW}}, \mathbf{i_{LR}}) \mid \Lambda \times (\mathbf{i_{LW}}, \mathbf{i_{LR}}, \mathbf{g}, 1)^T \geq \mathbf{0}\}$.

Each convex polyhedron $\lambda^{S_{LW} \to S_R}$ represents instances of statements that form a definition/dataflow use pairs. This polyhedron is constructed by enforcing the following conditions: *conflict condition* - write and read statement instances refer to the same memory location: $\mathcal{F}_{S_{LW}}(\mathbf{i_W}) = \mathcal{F}_{S_{LR}}(\mathbf{i_R})$, *causality condition* - a read instruction is scheduled after a write instruction: $\theta_{S_w}\mathbf{i_W} \prec \theta_{S_r}\mathbf{i_R}$, and *liveness condition* - there is no intervening write $w_k = < S_{KW}, \mathbf{i_{KW}} >$ happening between $< S_{LW}, \mathbf{i_{LW}} >$ and $< S_R, \mathbf{i_{LR}} >$) instruction instances.

Described polyhedra could be constructed by applying array dataflow algorithm detailed in [8]. Technical details on implementation of array dataflow algorithm inside GCC could be found in [17].

4.2 Algorithm

For the presentation purposes we use a property of GIMPLE three-address code stating that each statement can have at most one read or write to an array. This enables us to establish a one-to-one relationship between a statement and its memory access function.

In order to accomplish our violation-analysis based compilation flow, we have to provide two basic components in addition to what is already implemented in GRAPHITE: *array dataflow analysis* and *live range violation analysis*. Having those two components implemented, it is easy to set up an iterative, violation-analysis based polyhedral compilation flow as shown in Figure 10. In the rest of this subsection, we are providing some details about an implementation of those two components.

Array dataflow analysis Array dataflow analysis [8] essentially answers the following question: given a value v that is read, at some execution point, from a memory cell $M[b]$, compute the instruction instance $w =< S_{LW}, \mathbf{i_{LW}} >$ that is a source of value v.

We use a well known algorithm for array-dataflow analysis described in [8] and implemented in ISL library [22].

Array dataflow analysis considers only read-after-write (true) data dependences. Compared to a simple implementation of dependence analysis [20] which is currently used in GRAPHITE, it removes all transitively covered dependences.

The result of array-dataflow analysis is a set of dependence relations, summarized as dependence polyhedra: $\delta^{S_j \to S_i} = \{(< S_j, \mathbf{i_{LW}} >, < S_i, \mathbf{i_{LR}} >)\}$. Each dependence polyhedron $\delta^{S_j \to S_i}$ summarizes the dataflow relation between write/read statement instances, $< S_j, \mathbf{i_{LW}} > / < S_i, \mathbf{i_{LR}} >$ respectively.

Live range violation analysis After applying a transformation, the relative execution order of statement instances might change. This change might cause live ranges mapped to the same memory location to interfere.

The classical dependence analysis theory states that a source instruction instance must execute before a sink instruction instance: $\theta'_{S_{LW}}(\mathbf{i_{LW}}) \prec \theta'_{S_R}(\mathbf{i_{LR}})$. Once dataflow dependences are computed, we use a simple dependence violation analysis [20] to check this condition on all dataflow dependences.

Since we do not check for legality of memory based dependences, [8] how do we assure that a transformation would not force some statement instance, say $< S_{KW}, \mathbf{i_{KW}} >$, to overwrite a value v before it is read by an statement instance $< S_R, \mathbf{i_{LR}} >$ as it was expected in an original program?

In other words: we have to check that such a case may not happen after a transformation. We form the following system of equations, modelled using a polyhedral model:

$$Vio^{S_{LW} \to S_{KW} \to S_R} = \{(\mathbf{i_{LW}}, \mathbf{i_{LR}}) : \theta'_{S_{LW}}(\mathbf{i_{LW}}) \prec \theta'_{S_{KW}}(\mathbf{i_{KW}}) \prec \theta'_{S_R}(\mathbf{i_{LR}}) \wedge \\ \mathcal{F}_{S_{LW}}(\mathbf{i_W}) = \mathcal{F}_{S_{LR}}(\mathbf{i_R})\}$$

New schedules θ'_i represent a polyhedral transformation that we check for legality. If a violation set $Vio^{S_{LW} \to S_{KW} \to S_R}$ is empty, there is no violation after a transformation. We have to perform this check for each dataflow dependence polyhedron $\delta^{S_j \to S_i}$ and for each possible write statement S_{KW}.

[8] both write-after-write and write-after-read

Legality of loop parallelization: Previously mentioned check takes into an account the fact that that schedule functions θ'_{S_i} are describing a sequential execution order. Sequential execution order has a property that no two different statement instances could be scheduled at the same time: so either $\theta'_{S_i}(\mathbf{i}) \prec \theta'_{S_j}(\mathbf{j})$ or $\theta'_{S_j}(\mathbf{j}) \prec \theta'_{S_i}(\mathbf{i})$.

If we consider loop parallelization transformation, we need to take into an account that some statement instances might be executed simultaneously. Thus, value live range legality violation check used for sequential transformations could not be used.

Given a loop at level k that we consider for parallelization, we proceed as follows: we check that there are no *loop carried* dataflow dependences at level k. Then, we check that no two distinct statement instances $\theta'_{S_i}(\mathbf{i})$ and $\theta'_{S_j}(\mathbf{j})$ sharing the same prefix up to depth $k-1$ write to the same memory location (instances associated with the same iteration of the parallel loop but different iterations of some inner loops can still write to the same memory location).

Supporting array/scalar expansion: Our approach is compatible with well known array/scalar expansion approaches. If a transformation produces at least one violated live range (the set Vio is not empty) then we can choose to expand the variable M whose live ranges are violated. A precise characterization of violated live range instances in a set Vio could be used to drive the needed degree of expansion. Our proposed heuristic is to use the minimal sufficient degree of expansion so to correct all violated live ranges. If we do not want to perform an expansion, we can choose a new schedule that does not violate any live ranges.

Supporting privatization: Privatization is a common concept in the loop parallelization community. We can use our framework to automatically detect which scalars/arrays need to be privatized to enable loop parallelization transformation. This is actually how GRAPHITE is driving *autopar* – a GCC loop parallelizer.

4.3 An example

Let us take the GIMPLE code from Figure 5 and consider a memory location `phi_out_of_ssa`. Figure 11 shows an interleaving of writes and reads to this memory location. A slice of execution trace, for a finite number of iterations, is shown. Value live ranges are shown as well. Some value live range instances contained in a set: $(< S_1, (0,0) >, < S_2, (0,0,0) >)$, $(< S_6, (0,0,0) >, < S_2, (0,0,1) >)$, $(< S_6, (0,0,1) >, < S_2, (0,0,2) >)$, $(< S_6, (0,0,N-2) >, < S_2, (0,0,N-1) >)$, $(< S_1, (0,1) >, < S_2, (0,1,0) >)$.

After array dataflow analysis, we come up with two closed form expressions: $\lambda^{S_1 \to S_2}$ and $\lambda^{S_6 \to S_2}$. These two polyhedra summarize value live range instances between statements S_1 and S_2, and between S_6 and S_2 respectively. They have the following form:

$$\lambda^{S_1 \to S_2} = \{< (i,j), (i',j',k') >: i' = i \wedge j' = j \wedge k' = 0 \wedge 0 \le i < N \wedge 0 \le j < N\}$$

$$\lambda^{S_6 \to S_2} = \left\{ \begin{array}{l} < (i,j,k), (i',j',k') > \mid i' = i \wedge j' = j \wedge k' = k+1 \wedge 0 \le i < N \wedge \\ 0 \le j < N \wedge 0 \le k < N-1 \end{array} \right\}$$

For the purpose of this example, we would like to check whether interchanging loops i and j is a transformation that preserves non-conflicting condition on all live range instances. Referring again to Figure 5, we see that we are interested in the schedule of statements S_1, S_2, and S_6. Their scheduling functions in an original program are as follows: $\theta_{S_1}(i,j)^T = (0,i,0,j,0)^T$, $\theta_{S_2}(i,j,k)^T = (0,i,0,j,1,k,1)^T$, $\theta_{S_6}(i,j,k)^T = (0,i,0,j,1,k,8)^T$.

If we perform a loop interchange transformation, we will get the following transformed scheduling functions: $\theta'_{S_1}(i,j)^T = (0,j,0,i,0)^T$, $\theta'_{S_2}(i,j,k)^T = (0,j,0,i,1,k,1)^T$, $\theta'_{S_6}(i,j,k)^T = (0,j,0,i,1,k,8)^T$.

We construct the following violation sets: $Vio^{S_1 \to S_1 \to S_2}$, $Vio^{S_1 \to S_6 \to S_2}$, $Vio^{S_6 \to S_1 \to S_2}$, $Vio^{S_6 \to S_6 \to S_2}$, and after applying a sequential version of violation check, we find that those sets are empty, thus proving that there are no violations of live range intervals, even after an interchange transformation.

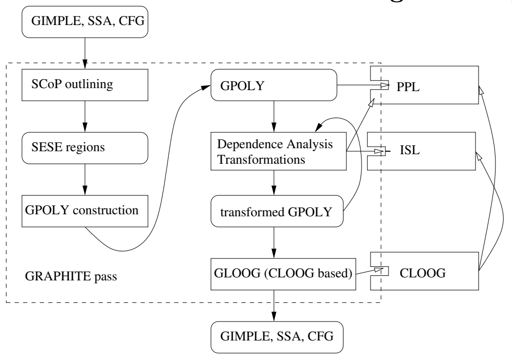
This check has to be performed for other memory accesses as well. In addition, we perform a dependence violation analysis on dataflow (read-after-write) dependences only: $\delta^{S_1 \to S_2}$ and $\delta^{S_6 \to S_2}$, stating that relative writes/reads are executed in a correct order.

As shown, a combination of dependence violation analysis of dataflow dependences and live range violation check reveals that it is legal to perform an interchange of i and j loops, even if the code was scalar optimized before entering GRAPHITE. Even if the code was not scalar optimized by compiler, an user might have written a code in such a way.

If we have used a dependence violation analysis of all dependences, including for example write-after-write dependence on `phi_out_of_ssa` (shown as $\delta^{WAW}_{S_1 \to S_1}$ in Figure 6), the dependence violation analysis would consider this transformation to be illegal, thus precluding (obviously) legal interchange.

5 Implementation

Fig. 12. Compilation flow



The GRAPHITE pass of GCC already has a dependence violation analysis, but it does not have live range interval violation analysis nor array dataflow analysis. The main task of GRAPHITE is to extract the *Static Control Parts* (SCoPs) amenable to polyhedral analyses and transformations, to build he polyhedral representation for each SCoP from its three-address GIMPLE representation, to perform various optimizations and analyses in the polyhedral model, and to regenerate the GIMPLE three-address code that corresponds to transformations in the polyhedral model. This four-stage process is the classical flow in polyhedral compilation of source-to-source compilers [11, 4]. Because the starting point of the GRAPHITE pass is the low-level three-address GIMPLE code instead of the high-level syntactical source code, some information is lost: the loop structure, loop induction variables, loop bounds, conditionals, data accesses and reductions. All of this information has to be reconstructed in order to build the polyhedral model representation of the relevant code fragment.

Figure 12 shows the four stages inside the current GRAPHITE pass: (1) the SCoPs are outlined from the control flow graph, (2) polyhedral representation is constructed for each SCoP (GPOLY construction), (3) data dependence analysis and transformations are performed (possibly multiple times), and (4) GIMPLE code corresponding to transformed polyhedral model is regenerated (GLOOG).

GRAPHITE is dependent on several libraries: PPL – Parma Polyhedra Library [2], CLooG – Chunky Loop Generator (which itself depends on PPL). Our algorithm would make GRAPHITE dependent on ISL [22] (Integer Set Library) as well. There is a special version of CLooG that is based on ISL library. More detailed explanation of the design and implementation of GRAPHITE can be found in [18].

For efficiency reasons, schedule and domain polyhedra are kept per each basic block, and not per each statement. This approach was taken in GRAPHITE to save memory and to reduce compilation time. Our approach requires that each statement would have a scheduling function. We still keep scheduling polyhedra per basic block only, and we provide the last component of the schedule for each statement on the fly.

This work is augmenting GRAPHITE dependence analysis with a more powerful dependence computation: it uses an array dataflow analysis instead of a simple memory access conflict check. A dataflow analysis is already implemented in ISL library, and we use this implementation in GRAPHITE.

GRAPHITE uses PPL as a library for the polyhedral representation of GIMPLE code. The latest version of PPL library includes an integer and mixed-integer linear programming algorithms [2]. ISL library [22] provides an implementation of the dataflow analysis algorithm used in this presentation. ISL library is linked into GCC and it is used to perform dataflow analysis. The routines that perform a transformation between ISL and PPL formats are implemented as well.

As a note, we want to mention that it is possible to implement array dataflow analysis algorithm in PPL library as well, and this is considered as a possible future work. Since PPL and ISL polyhedral libraries use different algorithms for polyhedra manipulations, a comparison of their scalability with respect to dataflow analysis algorithm might be interesting.

GRAPHITE is able to recognize loops that could be executed in parallel, by analyzing data dependences carried between loop iterations. If it finds that there are no loop carried dependences between loop iterations, it marks that loop as a parallel. This is done by setting a flag inside *loop* structure.

Parallel loop flag is in turn recognized by *autopar*, which is the loop parallelization pass in GCC. The main task of *autopar* pass is to distribute loop iterations into independent threads. This is accomplished by generating an appropriate calls to GOMP – an OpenMP runtime library.

Autopar is able to automatically discover scalar variables that have to be privatized – so that each thread would get its private copy. We use this property: when our analysis framework discovers live-range conflicts on scalar variables inside loops that are marked

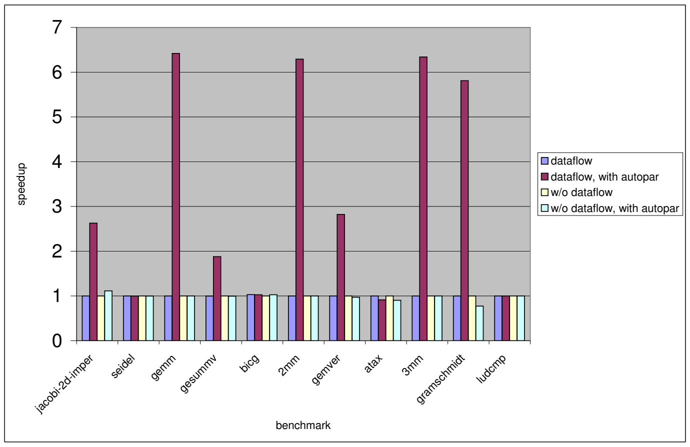
Benchmarks	**With Dataflow**		**Without Dataflow**	
	outer	inner	outer	inner
2mm.c	7	0	0	5
3mm.c	10	0	0	7
atax.c	1	2	0	3
bicg.c	1	1	0	2
gemm.c	4	0	0	3
gemver.c	4	1	0	3
gesummv.c	2	0	0	1
gramschmidt.c	3	1	0	4
jacobi-2d-imper.c	3	0	1	1
ludcmp.c	1	0	0	1
seidel.c	1	0	0	1

Table 1.

for parallelization, we have a guarantee that autopar can privatize them. This is not the case for memory conflicts on arrays: current implementation cannot privatize arrays.

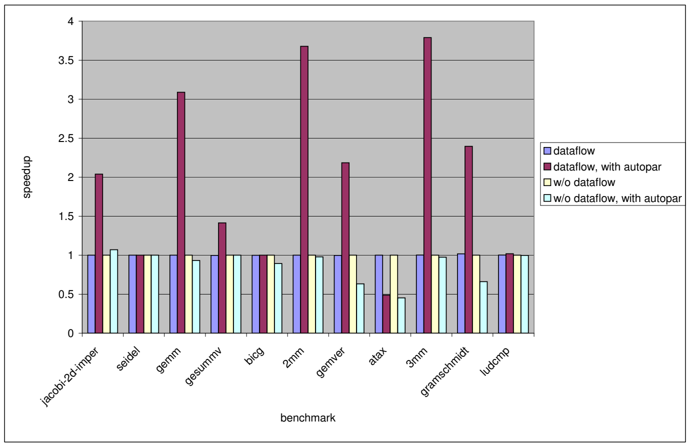
6 Results

Fig. 13. Speedups on X86 platform.



We ran measurements on a suite of computationally intensive numerical kernels. We compared the parallelizability of the kernels using both dependence models: the current, naive all data-dependences analysis, and the new array data-flow and live range analysis.

Fig. 14. Speedups on Power platform.



Two different machines were used to measure the speedup. The first was a Dell M600 blade with 2 quad core 2.66GHz Intel Xeon E5430 processors. The machine included 16GB RAM, 2x6 MB L2 cache per CPU, 32 KB I-cache and 32 KB D-cache L1 cache per core. The operating system kernel was Mandriva Linux with 2.6.29.3 kernel. The second was an IBM JS22 (7998-61X) blade with two dual-core 4.0 GHz POWER6 SCM processors. Each processing core supports two hardware threads. The machine includes 8 GB DDR2 RAM, 64 KB I-cache and 64 KB D-cache L1 cache per core, and 4 MB L2 cache per core. The operating system kernel was Linux 2.6.27 for PowerPC.

We found that the new violation analysis allowed for more loops – and typically more outer loops – to get parallelized, as depicted in Table 1.

For the parallel runs, we used autopar which was configured to spawn 8 threads, for both platforms.

6.1 Speedups on Power

The speedups on the Power platform are summarized in Figure 14.

The baseline is the *w/o dataflow* column, standing for a sequential run when the current transformation violation analysis is used. The *dataflow* column represents a sequential run when the new array data flow + live range analysis is used. As expected, changing the analysis doesn't influence the runtime of sequential runs, since we perform no sequential code transformation, which is why these columns show the same runtimes for all kernels. Enabling parallelization (autopar) on top of the current dependence analysis produces only a mild speedup on *jacobi-2d-imper*, while it does not impact or sometimes even slows down the others.

When autopar is enabled with the new data-flow and live range analysis, we achieve substantial speedups of up to 3.8x. We experience one degradation in *atax* kernel, which we relate to the overhead incurred by autopar, unrelated to which data dependence model we use, since it happens for both models.

6.2 Speedups on x86

The speedups on the x86-64 platform are summarized in Figure 13. The picture is very similar to the one shown for the Power platform: autopar using the current violation analysis provides small benefits on *jacobi-2d-imper*, and has no effect, or a negative effect on the rest of the kernels.

When autopar is enabled with the new data-flow and live range analysis, we achieve impressive speedups of up to 6.4x. We experience a similar behavior for *atax* as on the Power platform.

The overall scaling seems similar for both platforms , except for gramschmidt and gemm, which show a more significant speedup on x86 than on power. We have not yet investigated the architectural differences and other phenomena such as memory locality, register usage, synchronization costs and code generation (instruction scheduling), that play a roll in influencing the performance of parallalelized programs. An elaborated cost-model has to be provided to assess the various parallelization decisions' costs.

7 Future work

We have shown the approach to deal effectively with an array expansion and a privatization, especially useful when applied to automatic parallelization problem. We enable the *parallelism detection*, but we have not shown a way to *extract parallelism*. Indeed, some forms of parallelism are not immediately detectable without enabling transformations. What is more, even if there is an existing parallelism at the inner loop level, it is much better if an outer loop parallelism is extracted. The goal is to extract as much *coarse grained* parallelism as possible. For that purpose, one has to search for the sequence of loop transformations (loop interchange, loop skewing, loop shifting) to enable the outermost loop parallelism.

Several works have proposed heuristics and scheduling algorithms to enable outer loop parallelism. In addition, extracting maximal parallelism is very often not enough to get the best possible performance: indeed, memory hierarchy has to be taken into an account as well. Thus, only a tight coupling of parallelism extraction and *loop blocking* for memory locality improvements, as done in [4], could lead to a maximal performance on SMP machines.

Our goal is to implement and modify a transformation heuristic shown in [4] altogether with our *lazy* expansion approach. In that way, we hope to parallelize codes where outer loop parallelism could not be achieved without a transformations. Also, we want to perform inner loop parallelization for those codes where no parallelism is currently detected.

Thread-level parallelism is not the only form of parallelism that could be extracted. GCC has an already implemented automatic vectorization pass which is capable of vectorizing the inner and some outer loops. The automatic parallelization and vectorization passes are currently decoupled in GCC. Currently the pass ordering is fixed, thus the vectorization always happens before an automatic parallelization. The code generated by automatic vectorization pass has to be again analyzed from scratch by an automatic parallelization pass. This could lead to missed optimization opportunities.

Providing an unified view of those two transformations we might provide additional speedups due to combined vectorization and thread-level parallelism.

8 Conclusion

We presented a algorithmic framework to maximize the effectiveness of loop transformations while maintaining per-hardware-thread memory usage invariant. Unlike traditional array expansion and contraction techniques, dependence removal can be implemented with guaranteed memory footprint constraints. This framework is a promising trade-off between array expansion and the the degrees of freedom for affine transformation. Solving this problem is critical in compilers whose optimizations are based on three-address code, as is GIMPLE in GCC. Our experiments demonstrate its effectiveness in an experimental version of the GRAPHITE pass of GCC, using the ISL library and its implementation of array dataflow analysis via parametric integer linear programming.

Acknowledgment This work has been partly funded by the European FP7 project TERAFLUX id. 249013, `http://www.teraflux.eu`.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2001.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
3. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, Sept. 2004.
4. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *PLDI*, June 2008.
5. A. Cohen. Parallelization via constrained storage mapping optimization. In *Intl. Symp. on High Performance Computing (ISHPC'99)*, number 1615 in LNCS, pages 83–94, Kyoto, Japan, 1999.
6. A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. In *Euro-Par'99*, number 1685 in LNCS, pages 375–382, Toulouse, France, Sept. 1999. Springer-Verlag.
7. J.-F. Collard. The advantages of instance-wise reaching definition analyses in array (s)sa. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '98, pages 338–352, London, UK, 1999. Springer-Verlag.
8. P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.
9. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.
10. FSF. *GNU Compiler Collection (GCC) Internals Manual*, 2010. http://gcc.gnu.org/onlinedocs/gccint.
11. S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3):261–317, June 2006. Special issue on Microgrids.
12. W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, 1993.

13. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3):649–671, 1998.
14. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *ACM Symp. on Principles of Programming Languages*, pages 2–15, Charleston, South Carolina, Jan. 1993.
15. L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI*, June 2008.
16. W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Symp. on Programming Language Design and Implementation (PLDI'01)*, pages 232–242, 2001.
17. K. Trifunovic and A. Cohen. Enabling more optimizations in graphite: ignoring memory based dependences. In *GCC Summit*, Ottawa, Canada, October 2010.
18. K. Trifunovic and et al. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop*, October 2010.
19. K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, North Carolina, Sept. 2009.
20. N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344, 2006.
21. N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *PACT*, pages 292–304, 2007.
22. S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.
23. M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, Paris, 1996.
24. M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.