Enabling more optimizations in GRAPHITE: ignoring memory-based dependences

Konrad Trifunovic
INRIA Saclay
konrad.trifunovic@inria.fr

Albert Cohen
INRIA Saclay
albert.cohen@inria.fr

Abstract

Data-dependences need to be analyzed to guarantee the legality of a loop transformations and parallelization. But many dependences are spurious memory-based dependences: they are induced by storing values in the same memory location. Spurious dependences reduce the degrees of freedom in loop transformations and parallelization. The effective handling of spurious data-dependences in GIMPLE is essential for the effectiveness of polyhedral compilation in GCC.

We show that most memory-based-dependences induced by the gimplification can be ignored, rather than scalar/array expanded. Our method relies on an extension of the violated-dependence-analysis technique implemented in GRAPHITE. It has a minimal impact on compilation time and guarantees that we are not loosing any transformation opportunity compared to a source-to-source compilers. We will detail the algorithm, the current state of the implementation and the future plans.

1 Introduction

Loop nest optimization and parallelization are two of the most important program optimizations for performance on multicore architectures. Each modern compiler needs a careful implementation of those transformations in order to achieve efficiency on current architectures.

Choosing the most effective loop nest optimization and parallelization strategy is a huge and unstructured optimization problem that compiler has to face [9], [8], [3], [21], [20]. The well known approach to this problem is the polyhedral compilation framework [9] 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. Data-dependences put constraints on the relative ordering of read and write operations.

But many dependences are spurious memory-based dependences[1] : they 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 degrees of freedom available to express effective loop transformations and parallelization.

They could be removed by introducing new memory locations, i.e. *expansion* of the data structures [4]. While the *expansion* approaches might remove many spurious 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 intermediate code. Designing a polyhedral compilation framework on 3-address code exacerbates the problem of spurious memory dependences even further, since the gimplification process introduces many temporary variables.

In this paper, we show a technique guaranteeing that all the memory-based dependences induced by the lowering of a source program into GIMPLE can be ignored, rather than removed through scalar/array expan-

[1] anti and output dependences [1]

sion. This is excellent news to many loop transformation experts, as it circumvents a well known difficulty with polyhedral compilation techniques. Our method relies on an extension of the violated dependence analysis technique already implemented in Graphite.

2 State of the art

Spurious 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* [6] form, where each memory location is assigned only once.

Clearly, 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.

There are many works trying to find the best compromise between two extremes. They basically take two general approaches:

- Perform a maximal expansion, do a transformation, and then do an array contraction which minimizes the memory footprint. Approaches like [12], [11], [5] 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 [4], [14] 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 following strategy: we do not expand memory before scheduling. We simply ignore all memory based dependences, and we accept any proposed schedule. Only after, 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.

Figure 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];
    }
```

Figure 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;
    }
```

By taking our new approach, we are combining the best from two mentioned approaches: we do not perform a full expansion and we do not restrict the schedule. But there is a limitation to this: we are not able to compute schedules automatically by using linear programming approach as in [8], [3]. We must fall back to *iterative enumeration* of possible schedules as it is done in [16].

3 Motivating example

Consider a simple numerical kernel – the famous matrix multiplication – given in a Figure 1. A classical source-to-source polyhedral optimizer would see a simple static control loop with two statements only. A dependence graph is simple as well – it is shown in Figure 7. It contains both true (dataflow, read-after-write) dependences, and memory-based (write-after-write and write-after-read) dependences. The data dependence graph does not prevent loops 'i' and 'j' to be interchanged.

If we want to compile this source code in GRAPHITE, things become more complicated. After source code

Figure 3: GIMPLE and CFG as seen by Graphite

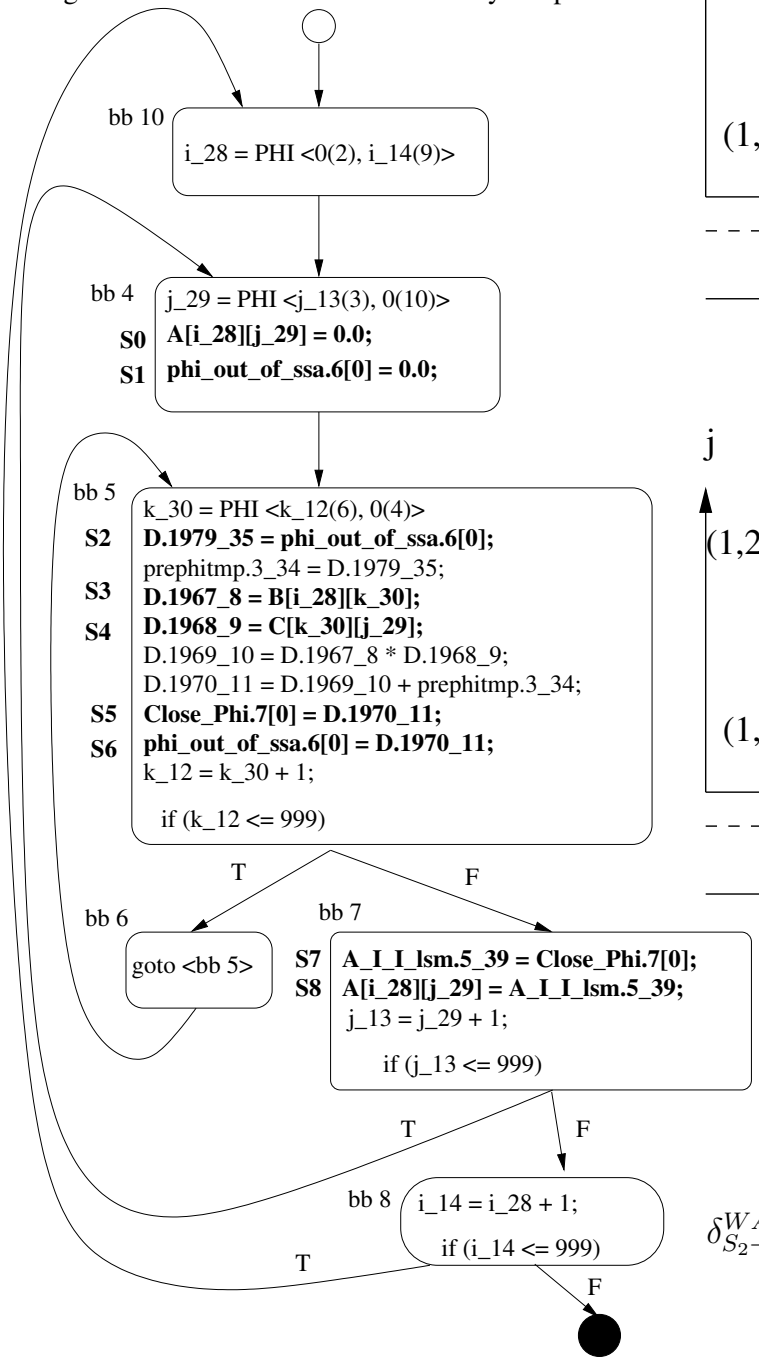


Figure 4: Legal execution order

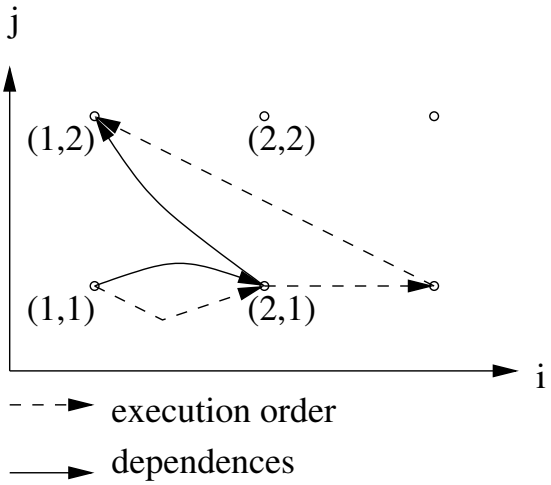


Figure 5: Illegal execution order

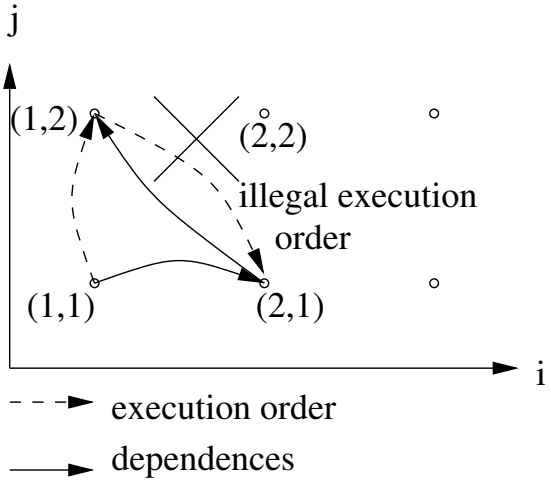


Figure 6: Data Dependence Graph

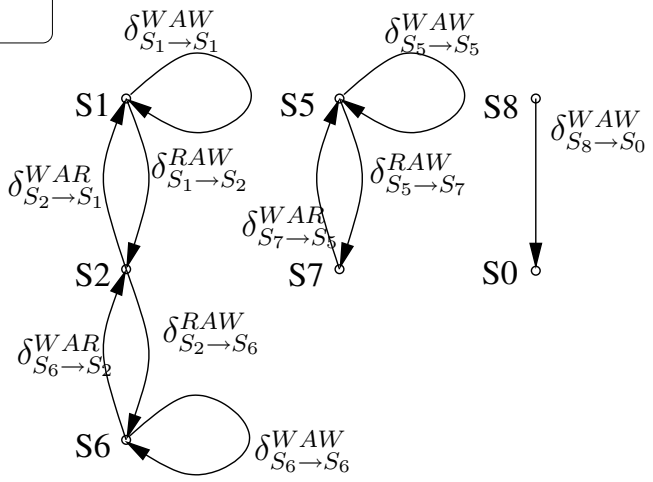
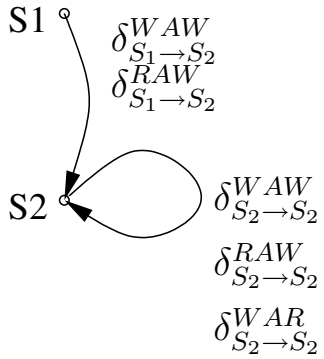


Figure 7: Matmult Data Dependence Graph



is transformed into GIMPLE it goes through many optimization 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 3.

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.

A dependence theory tells us that this dependence has to be respected. thus it enforces a sequential order on the code. Figure 4. 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 5. 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?

Figure 8: A flow of violation analysis based polyhedral compilation
INPUT: a SCoP

1. compute dataflow dependences
2. compute live range interval sets
3. choose a transformation
4. if dataflow dependence is violated
   (a) go to step 3.
5. if live range interval is violated
   (a) if we do not want to expand
      i. go to step 3.
   (b) if we want to expand
      i. perform an expansion of variable whose live range interval is violated, keep a schedule transformation and go to step 6.
6. generate the code

In the following sections we show how to formally prove which dependences could be ignored and which could not. We show an instance-wise variable live range analysis used to collect the information on memory usage patterns and violated dependence analysis used for checking whether a transformation destroys variable live ranges.

4 Framework

Polyhedral compilation traditionally takes as an input a dependence graph with all dependences, constructs a legal transformation using a mathematical framework (usually based on linear programming) and generates code.

The other class of so called *violation analysis* based compilation flow [18] takes as an input a dependence graph with all dependences as well, it constructs a transformation (without legality check), and only then it

checks whether the transformation is legal. If it is, then it proceeds with code generation. If it is not, then it reiterates and proposes a next transformation until it finds a legal one.

We take the violation analysis approach a step further: we do not take into the account all the dependences. We split the dependence graph into those dependences that are true *data-flow* dependences and those that are memory based.

When checking for legality of rescheduling, we assure that all true dependences are satisfied, and we do not check memory-based dependences for scheduling constraints. Instead, we construct *live range* sets for all the memory locations that are operated on. We check whether a transformation would destroy the live ranges. If not, then the transformation is legal.

If a transformation destroys live range set for a memory location, we could choose to expand those memory locations so as to repair the legality of live ranges, or we could abandon that transformation and choose another one. The choice on whether to expand could be based on a cost-model (what is a footprint) or it could be just a compilation parameter.

4.1 Some notation

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 [9].

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) [9, 3]. GIMPLE statements belonging to the SCoP should not contain calls to functions 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 [9, 3, 13]:

Iteration domains capture the 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: $\mathscr{D}_S = \left\{\mathbf{i} \mid \mathrm{D}_S \times (\mathbf{i},\mathbf{g},1)^T \geq \mathbf{0}\right\}$ 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).

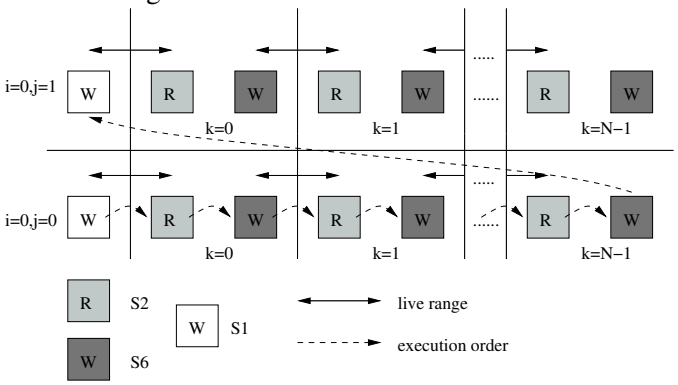
Data references capture the memory locations of array data elements on which GIMPLE statements operate. In each SCoP, by definition, the memory accesses are performed through array data references. A scalar variable can be seen as a zero-dimensional array. The data reference polyhedron $\mathscr{F}$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

Scheduling functions are also called scattering functions inside GRAPHITE following CLooG's 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) [8, 10]. This is done by constructing a *scattering polyhedron* representing the relation between iteration vectors and time stamp vector $\mathbf{t}$: $\theta = \left\{(\mathbf{i},\mathbf{t}) \mid \Theta \times (\mathbf{i},\mathbf{t},\mathbf{g},1)^T \geq \mathbf{0}\right\}$.

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

Figure 9: Read/Write instruction interleaving and variable live ranges



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 [17].

4.1.1 Live ranges

Execution trace of a sequential program can be seen as an interleaving of read and write instructions. This interleaving is encoded in a scheduling function of the polyhedral model, while memory accesses are encoded as data reference polyhedra.

We define a *live range* of a variable as the span of instructions in an execution trace between the first write(definition) of the variable and the last use (before it is killed). Given a GIMPLE code in Figure 3 we can model the execution trace and instances of live ranges as shown graphically in Figure 9.

Essentially, for a given scalar variable or memory cell in an array, there will be multiple instances of live ranges, since one scalar value might be overwritten and read multiple times inside some loop. Thus, we need a compact way to represent all instances of live ranges for each memory cell.

We use polyhedral representation to represent, in a compact manner, a set of instances of live ranges for a given memory location. Each instance of a live range is a tuple describing an instruction instance that is defining (writing) a value and an instance of last read instruction that is consuming (reading) a value before it is killed:

$$(< S_{LW}, \mathbf{i_{LW}} >, < S_{LR}, \mathbf{i_{LR}} >)$$

We consider a set of live range tuples:

$$L = \{(< S_{LW}, \mathbf{i_{LW}} >, < S_{LR}, \mathbf{i_{LR}} >)\}$$

This set represents an instancewise set of live ranges. We want to have a closed form expression that summarizes all the instances of live ranges.

We can decompose the set L into a set of convex polyhedra, each polyhedron describing live range instances for a pair of statements:

$$\lambda^{S_{LW} \rightarrow S_{LR}} = \{(\mathbf{i_{LW}}, \mathbf{i_{LR}}) : \Lambda \times (\mathbf{i_{LW}}, \mathbf{i_{LR}}, \mathbf{g}, 1)^T \geq \mathbf{0}\}$$

.

We define the set of live range instances for a given memory location M as the set of all non-empty live range instance polyhedra for each pair of statements that might form at least one live range:

$$\mathscr{L}_M = \{\lambda^{S_{LW} \rightarrow S_{LR}} : S_{LW}, S_{LR} \text{ start/end stmts of intervals}\}$$

Each convex polyhedron $\lambda^{S_{LW} \rightarrow S_{LR}}$ represents instances of statements that form a definition/last use pairs. This polyhedron is constructed by enforcing the following conditions:

Conflict condition: the definition statement instance and the last use statement instance refer to the same memory location: $\mathscr{F}_{S_{LW}}(\mathbf{i_W}) = \mathscr{F}_{S_{LR}}(\mathbf{i_R})$.

Causality condition: the read instruction is scheduled after write instruction: $\theta_{S_w}\mathbf{i_W} \prec \theta_{S_r}\mathbf{i_R}$.

Liveness condition: a live range is closed by a read instruction that is the latest read instruction before the variable is killed (by a subsequent write instruction). In polyhedral terms, this is expressed as:

$$\left\{ \begin{array}{l} \mathbf{i_{LR}} = lexmax(\mathbf{i_R}) \\ \wedge\, \mathbf{i_{LR}} \prec \mathbf{i_{KW}} = lexmin[\mathbf{i_W}](I : \mathbf{i_W} \prec I) \end{array} \right\}$$

The rest of this section will detail an algorithm for the computation of live ranges. All the necessary polyhedral operations are shown in details, so as to give a detailed implementation plan and a computational complexity estimate.

4.2 Algorithm details

There are four major algorithmic components we need to provide in order to support the idea of violation analysis based polyhedral compilation shown in Figure 8:

the array dataflow analysis algorithm is used to compute true(dataflow) dependences in a data dependence graph [7]. This information is used twice: in a violated dependence analysis check, and in a computation of memory live range intervals.

the memory live range interval analysis algorithm is used to compute sets of live range intervals for each memory location accessed in a SCoP. This information is used in the violated dependence analysis check to validate the transformation.

the live range violation analysis algorithm is used to check for the violation of live range intervals after a transformation. This check is the core of our new violated dependence analysis approach.

the dependence violation analysis algorithm [17] is already implemented in GRAPHITE. Currently it is used for checking the legality of both dataflow and memory-based dependences. In our new approach it is used for checking the violation of true dataflow dependences only, while the violation of memory-based dependences is replaced by the previously mentioned algorithm.

Each SCoP inside GRAPHITE is described as a collection of polyhedral components for each statement:

$$SCoP = \{< \mathscr{D}_{S_i}, \theta_{S_i}, \mathscr{F}_{S_i} >\}$$

For the presentation purposes, we will consider that we have scheduling functions kept independently for each statement. We also use a property of GIMPLE three address code, stating that each statement can have at most one read or write to an array.

We have additional attributes attached to the data reference access polyhedron: $base(\mathscr{F}_{S_i})$ returns the base address of the accessed array; $write(\mathscr{F}_{S_i})$, and $read(\mathscr{F}_{S_i})$ attributes have true value if the access is write/read respectively.

We use a $2 \cdot d + 1$ [9] encoding of the schedule timestamps. In the $2 \cdot d + 1$encoding, odd dimensions correspond to a *static schedule* – the precedence order of two statement instances that share the same loop, and are executed at the same iteration, is determined by their textual order inside that loop. Even dimensions correspond to *dynamic* schedule – if two statements share common loop, then the statement whose iteration comes earlier is executed before the other. There are as many even dimensions as the loop depth of the statement, hence the $2 \cdot d + 1$encoding. For example, schedules for statements S_1 and S_2 from Figure 3 are encoded in the following scheduling functions:

$$\theta_{S_1}(i,j)^T = (0,i,0,j,0)^T$$
$$\theta_{S_2}(i,j,k)^T = (0,i,0,j,1,k,1)^T$$

4.2.1 Array dataflow analysis

Array dataflow analysis [7] essentially gives a solution to the following question: for each scalar or array reference give the source instruction instance – an instruction instance that produced the value that reaches the given scalar or array reference. Array dataflow analysis considers read-after-write dependences only. Compared to a simple implementation of dependence analysis [17] currently used in GRAPHITE, it removes all transitively covered dependences.

The result is a list of dependence relations $\delta^{S_j \rightarrow S_i}$. Each dependence relation represents the relation between source and sink (write/read) iterations that access the same memory cell, so that the *read* access is getting the live value written in the *write* access, and not overwritten by any intermediate *write* access. The algorithm is outlined below:

$\forall S_i \in SCoP$ such that $read(\mathscr{F}_{S_i}) = T$ do:

1. $\forall S_j \in SCoP$ such that $[write(\mathscr{F}_{S_j}) = T$ and $base(\mathscr{F}_{S_j}) = base(\mathscr{F}_{S_i})]$ do:
   (a) $\mathbb{S}_W \leftarrow \mathbb{S}_W \cup S_j$

2. $depth \leftarrow 2 \cdot dim(\mathscr{D}_{\mathbb{S}_R}) + 1$

3. $I_{set} \leftarrow \mathscr{D}_{\mathbb{S}_R}$

4. for $lev \leftarrow depth$ to 1

5. $\forall S_j \in \mathbb{S}_W$ such that S_j can precede S_i at lev do:

   (a) $\{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\} = lexmax[\mathbf{i}_{\mathbf{R}} \in I_{set}](\mathbf{i}_{\mathbf{W}} : \mathscr{F}_{S_j}(\mathbf{i}_{\mathbf{W}}) = \mathscr{F}_{S_i}(\mathbf{i}_{\mathbf{R}}) \wedge \theta_{S_w}\mathbf{i}_{\mathbf{W}} \prec_{lev} \theta_{S_r}\mathbf{i}_{\mathbf{R}})$

   (b) $\delta^{S_j \rightarrow S_i} \leftarrow \delta^{S_j \rightarrow S_i} \cup \{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\}$

   (c) $I_{set} \leftarrow I_{set} \setminus range(\{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\})$

   (d) *call* **Remove killed sources**

Described in words, this algorithm does the following:

Iterate over all *read* accesses in a SCoP. Given a *read* access, iterate over all *write* accesses in a SCoP that write to the same memory cell (base addresses of arrays are the same). Compute for each iteration of the *read* access and for each array element accessed by that iteration, the *write* access that was the last to write the element accessed by the *read* access before this *read* access.

The set I_{set} keeps those iterations of the *read* access that are not yet processed. We proceed level by level, starting from the outermost level to the innermost. The core of the algorithm is the computation of the lexicographically maximal[2] iteration of the S_j statement (a write statement) that happens before the iteration of the S_i statement (a read statement).

That was the starting point of the computation: it computes a *possible* flow dependence from an instance of S_j statement, to an instance of S_i statement. But considering only one write/read pair is not enough: there might be some intermediate write instance of S_k statement that happens after an instance of the S_j statement, but before an instance of the S_i statement. That instance of the S_k statement is *killing* the value produced by an instance of S_j statement, since that value does not reach a read instance of the S_i statement. We take care of those cases in the **remove killed sources** procedure.

Procedure: **remove killed sources**

Given a possible flow dependence $\delta^{S_j \rightarrow S_i}$ at level lev, remove those elements for which there is an iteration of another source S_k that is closer to the sink S_i. Flow dependences $\delta^{S_k \rightarrow S_i}$ are updated with the improved sources, while flow dependences $\delta^{S_j \rightarrow S_i}$ are removed. Any improved source needs to precede the sink at the same level lev_{sink}, and needs to follow the source S_j at the same or a deeper level lev.

parameters: $SCoP$, $\mathbb{S}_W$, S_i, lev_{sink}, j

1. $depth \leftarrow 2 \cdot dim(S_j) + 1$

2. $\forall S_k \in \mathbb{S}_W, k < j$ such that S_k can precede S_i at lev_{sink} do:

   (a) for $lev \leftarrow lev_{sink}$ to $depth$

   (b) if S_j can precede S_k at lev

      i. $\{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\} = lexmax[\mathbf{i}_{\mathbf{R}}](\mathbf{i}_{\mathbf{KW}}, \mathbf{i}_{\mathbf{W}} : (\mathbf{i}_{\mathbf{W}}, \mathbf{i}_{\mathbf{R}}) \in \delta^{S_j \rightarrow S_i} \wedge \mathscr{F}_{S_k}(\mathbf{i}_{\mathbf{KW}}) = \mathscr{F}_{S_i}(\mathbf{i}_{\mathbf{R}}) \wedge \theta_{S_k}\mathbf{i}_{\mathbf{KW}} \prec_{lev_{sink}} \theta_{S_i}\mathbf{i}_{\mathbf{R}} \wedge \theta_{S_j}\mathbf{i}_{\mathbf{W}} \prec_{lev} \theta_{S_k}\mathbf{i}_{\mathbf{KW}})$

      ii. $\delta^{S_j \rightarrow S_i} \leftarrow \delta^{S_j \rightarrow S_i} \setminus \{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\}$

      iii. $\delta^{S_k \rightarrow S_i} \leftarrow \delta^{S_k \rightarrow S_i} \cup \{(\mathbf{i}_{LW}, \mathbf{i}_{\mathbf{R}})\}$

4.2.2 Memory live range interval analysis

What has been described so far is an algorithm to compute the latest write iteration, given a specific read iteration. In order to compute the set of live range instances of the variable $\mathscr{L}_M$, we also have to compute the latest sink iteration, given the source iteration. Thus, we apply a very similar procedure:

Procedure: **compute intervals** $\mathscr{L}_M$

for all distinct arrays M inside a SCoP do:

1. $\forall S_i$ such that $base(\mathscr{F}_{S_i}) = M$ and $write(\mathscr{F}_{S_i}) = T$

   (a) $\forall \delta^{S_i \rightarrow S_j}$ such that $source(\delta^{S_i \rightarrow S_j}) = S_i$

      i. $\mathbb{S}_R \leftarrow \mathbb{S}_R \cup sink(\delta^{S_i \rightarrow S_j})$

   (b) $\lambda^{S_i \rightarrow \mathbb{S}_R} \leftarrow$ compute the latest read for a write statement S_i and a set of read statements $\mathbb{S}_R$

   (c) $\mathscr{L}_M = \mathscr{L}_M \cup \lambda^{S_i \rightarrow \mathbb{S}_R}$

Given a set of already precomputed dataflow dependences $\delta^{S_i \rightarrow S_j}$, we are computing $\mathscr{L}_M$ sets for each distinct array M inside a SCoP (remember that scalars are also represented as arrays). For each statement S_i that

[2] It is a well known PIP (Parametric Integer Programming) algorithm implemented in polyhedral libraries such as PPL, ISL or PIPlib.

writes a value to an array M we collect all read statements for which the value is live. A set of those read statements is kept in $\mathbb{S}_R$. Among those read statements we compute the latest read access that reads a live value written in some instance of S_i statement. This is, as in a dataflow analysis, done by using a computation of lexicographic maximal iteration. For each write statement S_i we keep a result in $\lambda^{S_i \to \mathbb{S}_R}$, and we accumulate results to form a final set of live range intervals: $\mathscr{L}_M$.

4.3 Live range violation analysis

By definition, live range instances form a disjoint intervals in the original program execution trace. Dataflow (true, read-after-write) dependences enforce a correct execution order of statement instance pairs that produce and consume values respectively. If we enforce the correctness of all dataflow dependences and if we enforce that all live range intervals are preserved, we can guarantee a correctness of the transformation, ignoring the preservation of memory based dependences.

After applying a transformation, the relative execution order of statement instances might change. This change might induce a violation of live range interval instances. We say that two instances of live range intervals are *in conflict* if they overlap in the execution trace of a transformed program.

The goal of live range violation analysis is to check which instances, if any, of live range intervals are in conflict after a program transformation. If there is at least one pair of such instances, then the transformation is not legal without further corrections.

Computing live range interval conflict sets proceeds in two steps:

1. computing transformed image of live range interval set
2. checking for overlapping live range intervals

4.3.1 Computing transformed image of live range interval sets

Given an already computed initial set of live ranges $\mathscr{L}_M$ for an array M, we are interested in computing a transformed image of live ranges after applying a program transformation. This information is necessary for determining the legality of the transformation in a subsequent step.

If M is not a zero-dimensional array (a scalar represented as an array) then the polyhedron $\lambda^{S_{LW} \to S_{LR}}$ stores a family of live range sets for each array element. Live range sets for different memory locations might overlap in the execution trace of the original program. If we are interested in instances of live ranges for one particular memory location, then we have to parametrize that polyhedron with a vector of array indices $\mathbf{s}$ that identify an exact memory location that we are interested in:

$$\lambda^{S_{LW} \to S_{LR}}[\mathbf{s}] = \{(\mathbf{i_{LW}}, \mathbf{i_{LR}}) : \Lambda \times (\mathbf{i_{LW}}, \mathbf{i_{LR}})^T \geq \mathbf{0} \wedge \mathscr{F}_{S_{LR}}(\mathbf{i_{LR}}) = \mathbf{s}\}$$

In order to compute an image of transformed live range interval sets, we proceed with the following algorithm:

INPUT:

1. transformed schedules for the statements $\theta'_{S_1}, \theta'_{S_2}, \ldots, \theta'_{S_n}$
2. computed $\mathscr{L}_M$

$\forall \lambda^{S_{LW} \to S_{LR}} \in \mathscr{L}_M$ do:

1. $\lambda^{S_{LW} \to S_{LR}}[\mathbf{s}] \leftarrow extend(\lambda^{S_{LW} \to S_{LR}})$
2. $Image_M[\mathbf{s}] \leftarrow Image_M[\mathbf{s}] \cup \{(\mathbf{t_{LW}}, \mathbf{t_{LR}}) \mid \mathbf{t_{LW}} = \theta'_{S_{LW}}(\mathbf{i_{LW}}), \mathbf{t_{LR}} = \theta'_{S_{LR}}(\mathbf{i_{LR}}) \wedge (\mathbf{i_{LW}}, \mathbf{i_{LR}}) \in \lambda^{S_{LW} \to S_{LR}}[\mathbf{s}]\}$

An input to the algorithm is a program transformation, encoded as a set of transformed schedules for each statement, and the already computed $\mathscr{L}_M$ set. The algorithm proceeds by iterating over all polyhedra for different statement pairs that form live range interval sets: $\lambda^{S_{LW} \to S_{LR}}$. It extends each $\lambda^{S_{LW} \to S_{LR}}$ by parametrizing it with a vector of array indices $\mathbf{s}$. It computes an image for each $\lambda^{S_{LW} \to S_{LR}}[\mathbf{s}]$ by computing a time-stamp vector as an application of a scheduling function to an iteration vector. It accumulates partial results into $Image_M[\mathbf{s}]$ polyhedron. The output is a parametrized polyhedron $Image_M[\mathbf{s}]$ which holds a family of live range interval sets for each memory location identified by subscript vector $\mathbf{s}$.

4.3.2 Checking for overlapping live range intervals

The final building block needed for our framework is a live range violation analysis. We need to check whether any pair of live range intervals is in *conflict*.

We build a set of pairs of violated live range intervals *Vio*. A closed form expression to build a set of violated pairs is the following:

$$\begin{aligned} &Vio = \{< (\mathbf{t_{LW}}, \mathbf{t_{LR}}), (\mathbf{t_{LW}}', \mathbf{t_{LR}}') > | \\ &(\mathbf{t_{LW}}, \mathbf{t_{LR}}) \in Image_M[\mathbf{s}] \wedge \\ &(\mathbf{t_{LW}}', \mathbf{t_{LR}}') \in Image_M[\mathbf{s}] \wedge \mathbf{t_{LW}}' \prec \mathbf{t_{LR}} \wedge \mathbf{t_{LW}} \prec \mathbf{t_{LR}}'\} \end{aligned}$$

Please note that we use *lexicographic less than* operator $\prec$ when comparing time-stamps. This expression has to be evaluated level-by-level (for each time-stamp dimension), and the result is an union of polyhedra. The result can easily explode into the exponential number of polyhedra in the output. An upper bound to the number of polyhedra is $O(c^N)$. Luckily, parameter N is the loop depth inside SCoP, which is usually a small number ($N \leq 6$).

In the previous expression we used a property of *sequential schedules*: two different statement instances could not be scheduled at the same time – their time-stamps must differ. When checking starts/ends of overlapping intervals we do not need to check for the case where their time-stamps are equal, thus a strong *lexicographic less than* operator is enough.

Legality of parallelization: Previously mentioned property reduces the computational complexity when checking for a violation of sequential code transformations. If we consider loop parallelization transformation, then we need to take into the account that some statement instances might be executed at the same time, and their scheduling time-stamps might be equal.

A closed form expression to build a set of violated pairs in the case we want to check for parallelism transformation is the following:

$$\begin{aligned} &Vio = \{< (\mathbf{t_{LW}}, \mathbf{t_{LR}}), (\mathbf{t_{LW}}', \mathbf{t_{LR}}') > | \\ &(\mathbf{t_{LW}}, \mathbf{t_{LR}}) \in Image_M[\mathbf{s}] \wedge \\ &(\mathbf{t_{LW}}', \mathbf{t_{LR}}') \in Image_M[\mathbf{s}] \wedge \\ &\mathbf{t_{LW}}' \prec \mathbf{t_{LR}} \wedge \mathbf{t_{LW}} \prec \mathbf{t_{LR}}' \wedge \\ &(\mathbf{t_{LW}} \neq \mathbf{t_{LW}}' \vee \mathbf{t_{LR}} \neq \mathbf{t_{LR}}')\} \end{aligned}$$

This expression is computationally more heavy than the expression for sequential transformations, so it would be used only in the case where we check for legality of parallelism transformation.

Supporting array/scalar expansion: Our approach is compatible with well known array/scalar expansion approaches. If a transformation produces at least one pair of violated live range intervals (the set *Vio* is not empty) then we can choose to expand the variable M whose live range intervals 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 the violated live ranges. If we do not want to perform an expansion, we can choose a new schedule that does not violate any live range intervals.

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, or we can let the user specify (through OpenMP pragmas) which variables should be privatized. If some variable is explicitly marked as privatized, we need to modify access functions, so that we map a distinct memory location to each iteration.

4.4 An example

Let's take the GIMPLE code from Figure 3 and let's consider a memory location `phi_out_of_ssa`. Figure 9 shows an interleaving of writes and reads to this memory location. A slice of execution trace, for a limited number of iterations, is shown. Live ranges are shown as well.

Some live range interval instances contained in a set L:

$$\begin{aligned} &(< S_1, (0,0) >, < S_2, (0,0,0) >) \\ &(< S_6, (0,0,0) >, < S_2, (0,0,1) >) \\ &(< S_6, (0,0,1) >, < S_2, (0,0,2) >) \\ &(< S_6, (0,0,N-2) >, < S_2, (0,0,N-1) >) \\ &(< S_1, (0,1) >, < S_2, (0,1,0) >) \end{aligned}$$

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

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

$$\lambda^{S_6 \to S_2} = \{ < (i,j,k),(i',j',k') > | \; i' = i \wedge j' = j \wedge k' = k+1 \wedge 0 \leq i < N \wedge 0 \leq j < N \wedge 0 \leq k < N-1 \}$$

Two polyhedra are summarizing all instances of live range intervals for the location `phi_out_of_ssa`. We form a set of all polyhedra that describe live range interval set for a given location:

$$\mathscr{L}_M = \{\lambda^{S_1 \to S_2}, \lambda^{S_6 \to S_2}\}$$

We would like to check whether interchanging loops i and j is a transformation that preserves non-conflicting condition on all live range interval instances. Referring again to figure 3, 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:

$$\theta_{S_1}(i,j)^T = (0,i,0,j,0)^T$$
$$\theta_{S_2}(i,j,k)^T = (0,i,0,j,1,k,1)^T$$
$$\theta_{S_6}(i,j,k)^T = (0,i,0,j,1,k,8)^T$$

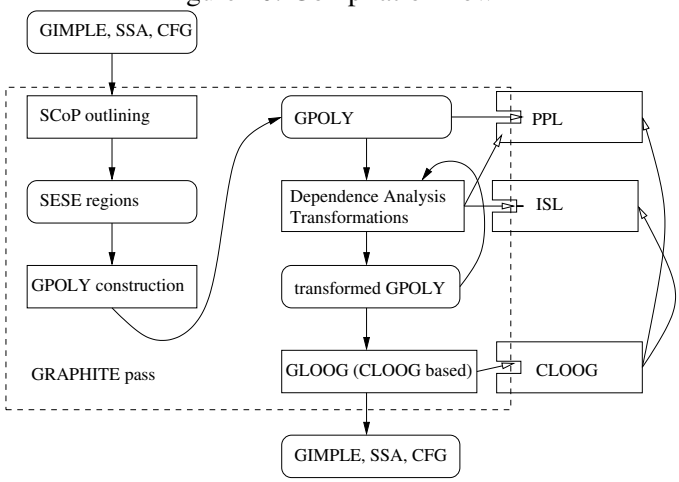
If we perform a loop interchange transformation, we will get the following transformed scheduling functions:

$$\theta'_{S_1}(i,j)^T = (0,j,0,i,0)^T$$
$$\theta'_{S_2}(i,j,k)^T = (0,j,0,i,1,k,1)^T$$
$$\theta'_{S_6}(i,j,k)^T = (0,j,0,i,1,k,8)^T$$

A transformed image of live range intervals is computed. The image is composed of an union of two polyhedra:

$$\{[(t1,t2,t3,t4,t5,t6,t7),(t1',t2',t3',t4',t5',t6',t7')] : t1 = t1' = 0 \wedge t2 = t2' \wedge t3 = t3' = 0 \wedge t4 = t4' \wedge \wedge t5 = 0, t5' = 1 \wedge t6' = 0, t7' = 1 \wedge 0 \leq t2 < N \wedge 0 \leq t4 < N \}$$

Figure 10: Compilation flow



$$\{[(t1,t2,t3,t4,t5,t6,t7),(t1',t2',t3',t4',t5',t6',t7')] : t1 = t1' = 0 \wedge t2 = t2' \wedge t3 = t3' = 0 \wedge t4 = t4' \wedge t5 = t5' = 1 \wedge t6' = t6+1 \wedge t7 = 8 \wedge t7' = 1 \wedge 0 \leq t2 < N \wedge 0 \leq t4 < N \wedge 0 \leq t6 < N-1 \}$$

Applying a sequential version of violation check on these polyhedra reveals that *Vio* set is empty, thus no intervals are conflicting. 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 $\delta^{S_2 \to S_6}$. A dependence violation analysis is already implemented in GRAPHITE.

As shown, a combination of dependence violation analysis of dataflow dependences and live range interval 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. If we use only dependence violation analysis of all dependences (dataflow and memory based) we will not be able to perform this transformation, since the dependence violation analysis would return

5 Implementation

We plan to implement our approach inside GRAPHITE pass of GCC compiler. GRAPHITE has an already implemented dependence violation analysis, but it does not have live range interval violation analysis nor array dataflow analysis. Those two algorithms were presented

in this paper and they would be implemented as components of GRAPHITE polyhedral compilation framework.

The polyhedral analysis and transformation framework called GRAPHITE is implemented as a pass in GCC compiler. The main task of this pass is to: extract the *polyhedral model* representation out of the GCC three-address GIMPLE representation, perform various optimizations and analyses on the polyhedral model representation and to regenerate the GIMPLE three-address code that corresponds to transformations on the polyhedral model. This three stage process is the classical flow in polyhedral compilation of source-to-source compilers [9, 3]. 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 10 shows stages inside current GRAPHITE pass: (1) the *Static Control Parts* (SCoP's) 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 [19] (Integer Set Library) as well. There is a special version of CLooG that is based on ISL library. More detailed explanation of GRAPHITE design and implementation internals is given in [15].

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, so the question is: how do we provide a scheduling function for each statement?

The answer is simple: since a basic block is a collection of statements, scheduling functions of all statements inside the basic block are the same, except the latest static scheduling component. Thus, we still keep scheduling polyhedra per basic block, 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 plan to use this implementation in GRAPHITE. This would require introducing a new polyhedral library in GCC, since we already use PPL library for internal polyhedral representation(GPOLY) inside GRAPHITE.

Several libraries for polyhedral operations are available: Polylib, PIPlib, PPL, ISL. Polylib and PIPlib are historically important, but not robust enough for production quality tools like GCC. GRAPHITE uses PPL as a library for the internal polyhedral representation of GIMPLE code. The latest version of PPL library includes an integer and mixed-integer linear programming algorithms, but one drawback of PPL library is that it is not an integer, but rational polyhedra library [2]. ISL library [19], on the other hand, provides integer solutions only. This is the key property needed in exact dataflow analysis. We have opted for ISL library, and we would propose to include this library as a requirement for GRAPHITE compilation.

Nevertheless, PPL would remain a standard interface for internal polyhedral operations inside GRAPHITE. It is used by CLooG code generator as well. Conversion operations between PPL and ISL polyhedra representation are already provided.

6 Conclusion

We have shown a framework to effectively approach the memory expansion vs transformation expressiveness problem. This is a necessary component of any compiler that wants to offer effective automatic parallelization and loop transformations. Solving this problem is even more critical in compilers whose optimizations are based on three-address code, as is GIMPLE in GCC.

We have shown a motivating example that justifies the effort we want to put into implementing the presented approach. A successful implementation of this algorithm, based on ISL library and ISL implementation of array dataflow analysis, would enable GRAPHITE to parallelize much wider class of loops.

Having an exact array dataflow analysis in GRAPHITE would be beneficial in many other cases as well, since it provides an exact information on the flow of *values*, instead of mere scheduling constraints, as given by current data dependence analysis.

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] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *PLDI*, June 2008.

[4] A. Cohen. Parallelization via constrained storage mapping optimization. In *Intl. Symp. on High Performance Computing (ISHPC'99)*, number 1615, pages 83–94, Kyoto, Japan, 1999.

[5] 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, September 1999. Springer-Verlag.

[6] J.-F. Collard. The advantages of reaching definition analyses in Array (S)SA. In *11þLanguages and Compilers for Parallel Computing*, number 1656 in LNCS, pages 338–352, Chapel Hill, North Carolina, August 1998. Springer-Verlag.

[7] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, February 1991.

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, December 1992. See also Part I, one dimensional time, 21(5):315–348.

[9] 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.

[10] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, 1993.

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

[12] 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, January 1993.

[13] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *PLDI*, June 2008.

[14] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *ACM Symp. on Programming Language Design and Implementation (PLDI'01)*, pages 232–242, 2001.

[15] K. Trifunovic and et al. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop*, October 2010.

[16] 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, September 2009.

[17] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344, 2006.

[18] Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *PACT*, pages 292–304, 2007.

[19] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010*, volume

6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.

[20] Michael E. Wolf, Dror E. Maydan, and Ding-Kai 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.

[21] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.