GRAPHITE Two Years After

First Lessons Learned From Real-World Polyhedral Compilation

Konrad Trifunovic[2], Albert Cohen[2], David Edelsohn[3], Li Feng[6], Tobias Grosser[5], Harsha Jagasia[1], Razya Ladelsky[4], Sebastian Pop[1], Jan Sjödin[1], and Ramakrishna Upadrasta[2]

[1] Open Source Compiler Engineering, AMD, Austin, Texas, USA
{harsha.jagasia, sebastian.pop, jan.sjodin}@amd.com
[2] INRIA Saclay – Île-de-France and LRI, Paris-Sud 11 University, Orsay, France
{albert.cohen, konrad.trifunovic, ramakrishna.upadrasta}@inria.fr
[3] IBM T. J. Watson Research, Yorktown Heights, USA
dje@watson.ibm.com
[4] IBM Haifa Research, Haifa, Israel
razya@il.ibm.com
[5] University of Passau, Passau, Germany
grosser@fim.uni-passau.de
[6] Xi'an Jiaotong University, Xi'an, China
nemokingdom@gmail.com

Abstract. Modern compilers are responsible for adapting the semantics of source programs into a form that makes efficient use of a highly complex, heterogeneous machine. This adaptation amounts to solve an optimization problem in a huge and unstructured search space, while predicting the performance outcome of complex sequences of program transformations. The polyhedral model of compilation is aimed at these challenges. Its geometrical, non-inductive semantics enables the construction of better-structured optimization problems and precise analytical models. Recent work demonstrated the scalability of the main polyhedral algorithms to real-world programs. Its integration into production compilers is under way, pioneered by the GRAPHITE branch of the GNU Compiler Collection (GCC). Two years after the effective beginning of the project, this paper reports on original questions and innovative solutions that arose during the design and implementation of GRAPHITE.

1 Introduction

Despite several decades of research into the Polyhedral model, there is still no general-purpose production compiler using the Polyhedral model internally. The situation is changing with the demonstration of the scalability of polyhedral algorithms and with the widespread dissemination of multicore processors and hardware accelerators. Two proprietary polyhedral compilers are in development: the R-Stream compiler from Reservoir Labs [26], and IBM's polyhedral extension of its XL compiler suite [35].

This paper describes the GRAPHITE compilation pass of GCC, embedding polyhedral analyses and transformations into the GNU Compiler Collection (GCC) [32]. Polyhedral information is extracted directly from the GIMPLE intermediate representation, in three-address, *Static Single Assignment* (SSA) form. This is a major difference with traditional source-to-source polyhedral compilers which operate on high-level abstract syntax. Operating directly on the three-address code brings in new challenges but also new opportunities: we can leverage existing analyses in the compiler and interact with a wealth of optimizations.

GRAPHITE is based on the polyhedral representation designed by Girbal et al. [13]. This rich algebraic representation enables the composition of polyhedral generalizations of classical loop transformations, decoupling them from the syntactic form of the program. Classical transformations like loop fusion or tiling can be composed in any order and generalized to imperfectly-nested loops with complex domains, without intermediate translation to a syntactic form (avoiding code size explosion). GRAPHITE also aims at providing precise performance models and profitability prediction heuristics. Its applications include automatic parallelization and vectorization, offloading of computational kernels onto hardware accelerators, memory hierarchy usage optimizations, cost modelling and static code analysis (e.g., static debugging of parallel programs).

The paper is structured as follows. Section 2 discusses related work. Section 3 describes the design of GRAPHITE. Section 4 explores the current optimizations implemented in GRAPHITE. Representation of pointer accesses is an original issue for polyhedral compilation and is presented in Section 5, before the conclusion in Section 6.

2 Related work

There have been many efforts in designing an advanced loop nest transformation infrastructure. Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope [10] and Polaris [6] are dependence based, source-to-source parallelizers for Fortran. KAP [19] is closely related to these academic tools.

SUIF [16] is a platform for implementing advanced compiler prototypes. PIPS [17] is one of the most complete loop restructuring compiler, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). Both of them use a syntax tree extended with polyhedral annotations, but not a unified polyhedral representation.

The MARS compiler [28] unifies classical dependence-based loop transformations with data storage optimizations. However, the MARS intermediate representation only captures part of the loop information (domains and access functions): it lacks the characterization of iteration orderings through multidimensional affine schedules.

The first thorough application of the polyhedral representation was the Petit tool [20], based on the Omega library [23]. It provides space-time mappings for iteration reordering, and it shares our emphasis on per-statement transformations, but it is intended as a research tool for small kernels only. We also use a code generation technique that is again significantly more robust than the code generation in Omega [4].

Semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [21, 9, 39, 8, 36]. They do not define automatic methods or integrate a model-based heuristic to construct profitable optimization strategies.

The GRAPHITE project was first announced by Pop et al. in 2006 [32] but real development work started only one year later: the number of changes committed to the GRAPHITE branch are presented in Figure 2. The design of GRAPHITE is largely borrowed from the WRaP-IT polyhedral interface to Open64 and its URUK loop nest optimizer [13]. The CHiLL project from Chen et al. revisited the URUK approach focusing on source-to-source transformation scripting [8, 36]. Unlike URUK and CHiLL, GRAPHITE aims at complete automation, possibly resorting to iterative search or statistical modeling of the profitability of program transformations. Besides, unexpected design and implementation issues have arisen, partly due to the design of GCC itself, but mostly due to the integration of the polyhedral representation in a general-purpose compilation flow, such as pointers, profile data, debugging information, resource usage (compilation time), pass ordering, interaction among passes, etc.

3 Design

The polyhedral analysis and transformation framework called GRAPHITE is implemented as a pass in the GNU Compiler Collection compiler. The main task of this pass is to: extract the *polyhedral model* representation out of the GCC three-address GIMPLE representation, perform the 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 [13, 7]. 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.

Year	Commits
2009	497
2008	216
2007	36
2006	10

Fig. 2. Changes to the GRAPHITE branch

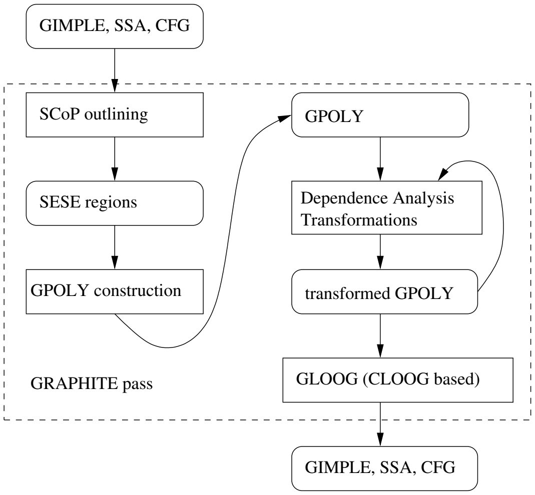


Fig. 1. Stages of the GRAPHITE pass

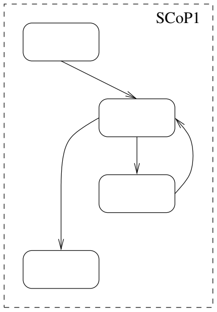


Fig. 3. Full SCoP

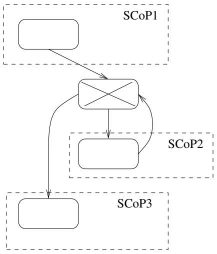
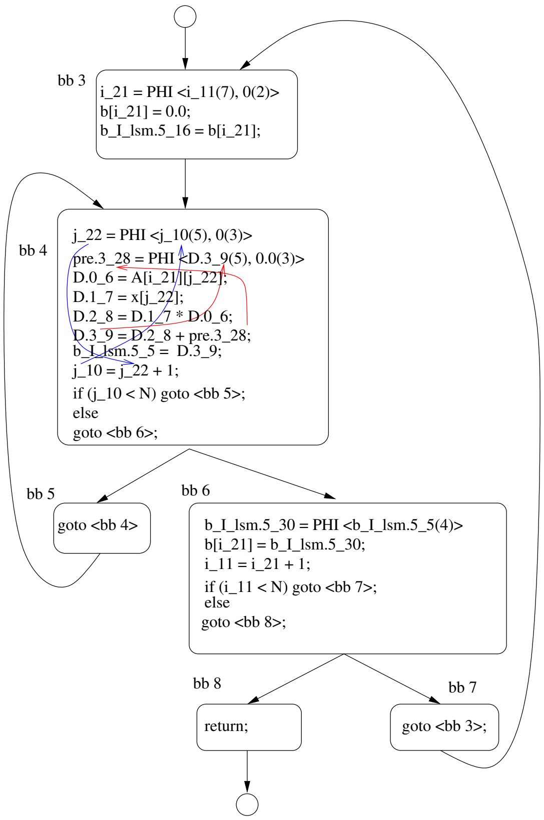


Fig. 4. Split SCoP

Figure 1 shows the stages inside the GRAPHITE pass: (1) the *Static Control Parts* (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). The details of each stage are given in the following subsections.



```
i_21 = PHI <i_11(7), 0(2)>
b[i_21] = 0.0;
b_I_lsm.5_16 = b[i_21];
```

```
j_22 = PHI <j_10(5), 0(3)>
pre.3_28 = PHI <D.3_9(5), 0.0(3)>
D.0_6 = A[i_21][j_22];
D.1_7 = x[j_22];
D.2_8 = D.1_7 * D.0_6;
D.3_9 = D.2_8 + pre.3_28;
b_I_lsm.5_5 =  D.3_9;
j_10 = j_22 + 1;
if (j_10 < N) goto <bb 5>;
else
goto <bb 6>;
```

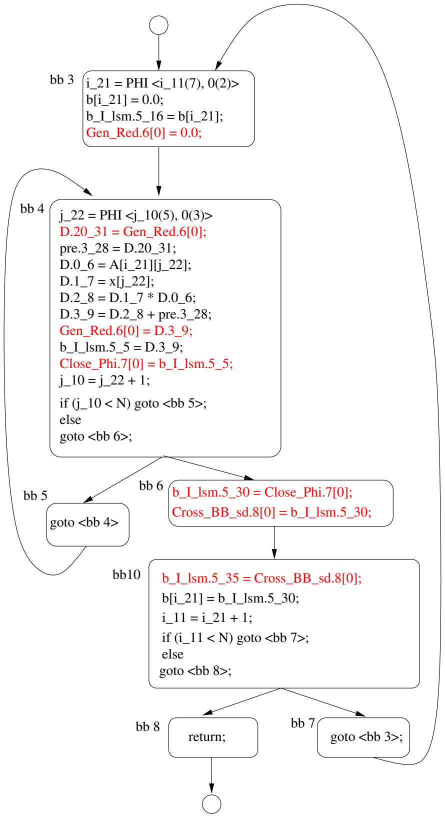
```
goto <bb 4>
```

```
b_I_lsm.5_30 = PHI <b_I_lsm.5_5(4)>
b[i_21] = b_I_lsm.5_30;
i_11 = i_21 + 1;
if (i_11 < N) goto <bb 7>;
else
goto <bb 8>;
```

```
return;
```

```
goto <bb 3>;
```

Fig. 5. GIMPLE code with CFG.



```
i_21 = PHI <i_11(7), 0(2)>
b[i_21] = 0.0;
b_I_lsm.5_16 = b[i_21];
Gen_Red.6[0] = 0.0;
```

```
j_22 = PHI <j_10(5), 0(3)>
D.20_31 = Gen_Red.6[0];
pre.3_28 = D.20_31;
D.0_6 = A[i_21][j_22];
D.1_7 = x[j_22];
D.2_8 = D.1_7 * D.0_6;
D.3_9 = D.2_8 + pre.3_28;
Gen_Red.6[0] = D.3_9;
b_I_lsm.5_5 = D.3_9;
Close_Phi.7[0] = b_I_lsm.5_5;
j_10 = j_22 + 1;
if (j_10 < N) goto <bb 5>;
else
goto <bb 6>;
```

```
goto <bb 4>
```

```
b_I_lsm.5_30 = Close_Phi.7[0];
Cross_BB_sd.8[0] = b_I_lsm.5_30;
```

```
b_I_lsm.5_35 = Cross_BB_sd.8[0];
b[i_21] = b_I_lsm.5_30;
i_11 = i_21 + 1;
if (i_11 < N) goto <bb 7>;
else
goto <bb 8>;
```

```
return;
```

```
goto <bb 3>;
```

Fig. 6. Single-element arrays inserted to handle scalar dependences and reductions

Listing 1.1. Matvect

```
for (i = 0; i < N; i++) {
  b[i] = 0;
  for (j = 0; j < N; j++)
    b[i] += A[i][j] * x[j];
}
```

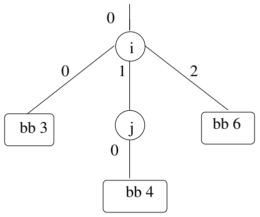


Fig. 7. LST tree

$$\mathcal{D}^S_{bb3} = \big\{(i) \mid 0 \le i \le N-1\big\}$$
$$\mathcal{D}^S_{bb4} = \big\{(i,j) \mid 0 \le i \le N-1 \land 0 \le j \le N-1\big\}$$
$$\mathcal{D}^S_{bb6} = \big\{(i) \mid 0 \le i \le N-1\big\}$$
$$\mathcal{F}_{dr1} = \big\{(i,a,s_1) \mid a = 0 \land s_1 = i \land 0 \le s_1 \le N-1\big\}$$
$$\mathcal{F}_{dr2} = \big\{(i,j,a,s_1) \mid a = 1 \land s_1 = j \land 0 \le s_1 \le N-1\big\}$$
$$\mathcal{F}_{dr3} = \big\{(i,j,a,s_1,s_2) \mid a = 2 \land s_1 = i \land s_2 = j \land 0 \le s_1, s_2 \le N-1\big\}$$
$$\mathcal{F}_{dr4} = \big\{(i,a,s_1) \mid a = 0 \land s_1 = i \land 0 \le s_1 \le N-1\big\}$$
$$\theta_{bb3} = \big\{(i,t_1,t_2,t_3) \mid t_1 = 0 \land t_2 = i \land t_3 = 0\big\}$$
$$\theta_{bb4} = \big\{(i,j,t_1,t_2,t_3,t_4,t_5) \mid t_1 = 0 \land t_2 = i \land t_3 = 1 \land t_4 = j \land t_5 = 0\big\}$$
$$\theta_{bb6} = \big\{(i,t_1,t_2,t_3) \mid t_1 = 0 \land t_2 = i \land t_3 = 2\big\}$$

Fig. 8. Components of the polyhedral representation of the GIMPLE code

3.1 SSA based SCoP outlining

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.

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) [13, 7]. 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.

Since the GRAPHITE pass is scheduled at the stage where three-address code is in Static Single-Assignment form, all the analyses based on the SSA are available for use in GRAPHITE. This is crucial: in order to perform SCoP outlining the *scalar evolution* analysis framework of GCC is used [33]. Scalar evolution relies on SSA form to compute closed form expressions for induction variables. These closed forms are represented by structures called *CHains of RECurrences* (CHREC).

Chains of recurrences might represent induction variables (loop induction variables, array access subscript functions) that are affine or non-affine. For example, the scalar evolution of the `j_22` variable in the basic block 4 (Figure 5) is expressed as follows: $\{0, +, 1\}_2$, meaning that the starting value of the induction variable is 0, and it is incremented by 1 in each iteration of the loop number 2 (loop corresponding to basic blocks 5 and 6).

SCoP outlining proceeds as follows: first, a new SCoP region is opened, and then the basic blocks of the CFG are scanned in the dominator order. If the basic block contains a statement that is not representable in the polyhedral model then the whole basic block is deemed *difficult* so the current SCoP region is closed and a new SCoP is opened at the basic block dominated by the *difficult* basic block. Figure 3 shows one SCoP containing all the basic blocks, whereas Figure 4 shows how the *difficult* statement causes multiple SCoPs to be formed.

There exist limitations to the SCoP detection algorithm currently implemented in GRAPHITE: for example, determining whether the scalar evolution of a variable could be handled in the polyhedral model, or whether the variable should be considered a parameter of the SCoP. We think that a detection of SCoPs based on the structured CFG with SESE regions [18] would be more appropriate. A structural SCoP detection traverses the regions tree starting from the outermost SESE region, and tries to prove that all the statements in that maximal region can be handled in the polyhedral representation. When a *difficult* statement is detected, the statement is analyzed in all the regions containing it, from the outermost region to the innermost one, until either the statement is simple enough in a smaller region, or the region is the statement itself, in which case the statement cannot be handled at all. We consider the integration of a structural SCoP detection algorithm in future versions of GCC.

3.2 Construction of the polyhedral representation

Once the SCoPs are outlined, the polyhedral information is built for each basic block contained in a SCoP. The polyhedral representation consists essentially of three components: iteration domains, schedules, and data accesses.

The polyhedral information attached to each basic block in a SCoP is internally called GPOLY. All the components of the polyhedral model are represented as a system of affine equalities or inequalities, and for that purpose a *polyhedral library* is used. Currently, the Parma Polyhedra Library (PPL) [3] is used, but the representation is designed to accommodate other similar libraries.

Once again, the *scalar evolution* analysis framework is used to deduce the affine form of the loop bounds and global parameters (to build the iteration domains), memory addressing expressions (for the data accesses). Initial scheduling functions for each basic block are deduced from the *Loop Statement Tree* (LST) showing the relative ordering of the basic blocks, and initial nesting structure of the loops. An example of the LST is given in Figure 7. More details explaining all the components of the polyhedral model are given in the following subsection.

Contrary to source-to-source polyhedral compilers [17, 31, 25], we have chosen to represent the schedules and the domains on a per *Basic Block* (BB) granularity instead of on a per statement granularity. This choice is somewhat rigid, since it prevents the independent scheduling of GIMPLE statements belonging to the same basic block. On the other hand, constructing the polyhedral representation per basic block might provide greater scalability. SCoP control and data flow are represented with three components of the polyhedral model [13, 7, 34]:

Iteration domains capture the dynamic instances of all basic blocks — all possible values of surrounding loop induction variables — through a set of affine inequalities. Each dynamic instance of a basic block 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. The dimension of iteration vector $\mathbf{i}$ is d^S. If the basic block S belongs to a SCoP then the set of all iteration vectors $\mathbf{i}$ relevant for S can be represented by a polytope: $\mathcal{D}^S = \left\{\mathbf{i} \mid \mathrm{D}^S \times (\mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$ 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 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 (array with only one dimension and only one element `A[0]`). Each array data reference (`data_reference`) inside a basic block is wrapped inside a `poly_dr` structure which contains the *data reference polyhedron*. The data reference polyhedron $\mathcal{F}$ encodes the *access relation* mapping iteration vectors in $\mathcal{D}^S$ to the array subscripts represented by the vector $\mathbf{s}$: $\mathcal{F} = \left\{(\mathbf{i}, a, \mathbf{s}) \mid \mathrm{F} \times (\mathbf{i}, a, \mathbf{s}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$. The alias set number a captures points-to information (pointer aliasing); it allows to represent accesses through arbitrary pointers and will be defined in Section 5.

In contrast to classical polyhedral model representation [11, 22] we have chosen to represent *data references as relations*. This means that there is no one-to-one correspondence between iteration vector and subscript. This enables us to represent *memory regions* – when the data reference information is not complete (coming from interprocedural analysis for example). Nevertheless, very often, the correspondence between iterator vectors and data reference subscripts is a functional affine mapping $\mathbf{s} = f(\mathbf{i}, \mathbf{g})$.

In previous literature, the problem of link between dependence analysis and the analyses preceeding it, like alias analysis, has not been explored. This leads to inefficiency and impreciseness in representation, which are further exacerbated by software engineering constraints like modularity and portability. In Section 5, we will see a discussion of the problem and its algorithmic characterization as a hard combinatorial problem.

Scheduling functions are also called scattering functions inside GRAPHITE following CLooG's terminology. While iteration domains define the set of all dynamic instances for each basic block, 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) [11, 22]. This is done in GRAPHITE by constructing the *scattering polyhedron* representing the relation between iteration vectors and time stamp vector $\mathbf{t}$: $\theta = \left\{(\mathbf{t}, \mathbf{i}) \mid \Theta \times (\mathbf{t}, \mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$.

Dynamic instances are executed according to the lexicographical ordering of timestamp vectors. By changing the scattering function, we can reorder the execution order of dynamic iterations, thus performing powerful *loop transformations*. More details on the transformations are given in Subsection 3.5.

Given the example GIMPLE code in Figure 5, the components of the polyhedral model representation are given in Figure 8.

3.3 Dependence analysis

In order to represent the semantics of the original program in the polyhedral model the dependence between dynamic instances of statements needs to be represented. The dependences are necessary to guarantee the correctness of loop transformations.

We are considering *data dependences* coming from the reads and writes of array elements. By definition [38], there is a data dependence from the dynamic instance of a basic block $(S_i, \mathbf{i}_{S_i})$ to the dynamic instance of basic block $(S_j, \mathbf{i}_{S_j})$ if both iteration vectors belong to their respective iteration domains (the execution is feasible), both instances refer to the same memory location and at least one of the data references is write and the instance $(S_i, \mathbf{i}_{S_i})$ is executed before $(S_j, \mathbf{i}_{S_j})$.

The *Polyhedral Dependence Analysis* (PDA) implemented in GRAPHITE is an *instance-wise dependence analysis* – meaning that the dependences are represented as polyhedra encoding the dependence relations between basic block instances. If projected to the Cartesian product of two iteration domains [38, 7, 34], the polyhedron encodes the iteration of the source of the dependence and the iteration of the sink of the dependence:

3.4 Handling scalar dependences

While the classical dependence analysis in the source-to-source polyhedral compilers considers only the data dependences between arrays (treating scalars as zero-dimensional arrays), this approach is not the most appropriate in the context of three-address code in the SSA form. If we are not considering scalar dependences, we are not capturing all the semantical constraints of the program – the transformed code could be illegal. If we are to convert all scalars to zero-dimensional arrays we would greatly increase the compilation time (polyhedral dependence check is algorithmically costly) and produce inefficient code.

The approach taken in GRAPHITE framework is to classify the scalar dependences into the following categories:

Intra basic block dependences occur between scalars inside a basic block. Those dependences are not considered by PDA in GRAPHITE. Since the statements inside the basic block cannot be rescheduled, the scalar dependences between statements inside a same basic block are not affected by polyhedral transformations. Those dependences are captured by use-def chains of the SSA representation.

Cross basic block dependences occur between scalars belonging to two different basic blocks. Those scalars are rewritten into zero-dimensional (single-element) arrays, such that PDA considers them as the regular array accesses. An example is given in Figure 6, where new zero-dimensional arrays (called `Cross_BB_sd`) are introduced.

Reduction dependences occur in data flow cycles that contain associative and commutative operations, like an accumulator variable performing a summation over the values of an array. For the regular reductions the new zero-dimensional arrays are introduced (as seen in Figure 6, where `Gen_Red` and `Close_Phi` arrays are introduced). If the reduction operator can be proved to be commutative and associative, then the dependences are marked as belonging to such a reduction. The former enables the optimizations, since the reduction operations can be rescheduled, disregarding the data dependences, if proved to be associative and commutative.

3.5 Transformations

According to the compositional approach of polyhedral transformations [13], the composition of multiple loop transformations in the polyhedral model can be expressed as a single scheduling transformation. By modifying the scheduling relations θ for each basic block, and regenerating the GIMPLE code according to those new schedules, we are able to perform arbitrary rescheduling of the basic blocks inside a SCoP.

In order to preserve the legality of the transformations, the *legality check* is performed for each data dependence relation.

Given the original data dependence relation $\mathcal{P}_{(S_i,R_k)\ \rightarrow(S_j,R_l)}$ representing the pairs of iterations which need to be executed in the specific order, the other polyhedron $\mathcal{P'}_{(S_i,R_k)\ \rightarrow(S_j,R_l)}$ is computed, giving those pairs of iterations that are violating the original dependence (they are executed in reversed order according to the new schedule). If the intersection of two polyhedra is not empty, then there exists at least one pair of iterations that is executed in the wrong order, thus rendering the transformation *illegal*. The whole process is called *Violated Dependence Analysis* [38].

The task of GRAPHITE is to look for such transformations that are beneficial for optimizing various criteria, but which are legal at the same time. The simple search heuristic is looking for good transformations, rejecting those which are illegal. This is certainly an iterative process. If none of the transformations seems legal, then no transformation is done. GRAPHITE currently implements loop interchange, loop stripmining, loop distribution and loop-blocking.

3.6 Code generation

In source-to-source polyhedral compilers, the code generation pass is the last one, generating the new loop structures to scan statement instances in the order defined by the modified schedule.

In GRAPHITE it is not the syntactical source code that is the final result of the pass: GRAPHITE should be able to regenerate the GIMPLE code. Furthermore, the generated GIMPLE code has to be reinserted back into the CFG, respecting the SSA form invariants and passed to the further passes after GRAPHITE.

Multiple loop generation tools exist that operate on the polyhedral model. The most mature one is the CLooG (Chunky LOOp Generator) [4]. CLooG is used in GRAPHITE as the major component of code generation. Since CLooG is meant for generating syntactic code (mainly `C` code), it cannot be used directly: CLooG generates an internal representation called CLAST which is a simple abstract-syntax tree containing only loops, conditions, and statements. In our case statements are replaced with basic blocks.

CLooG is fed by the polyhedral representation (GPOLY) and is asked to generate a CLAST. The nodes of the abstract-syntax tree are pointers to original basic blocks. Depending on the loop transformations, the basic blocks might be rescheduled, moved to other loops, or even replicated (when performing a transformation). The final effect is represented in the CLAST. The CLAST tree is traversed and the basic blocks are put into the their new positions in the GIMPLE CFG, loop structures are regenerated and some basic blocks are replicated.

Even in the case of the identity transformation (no schedule modification), the newly generated loops according to the CLAST tree have the new induction variables. All the basic blocks belonging to a SCoP have to be scanned, and the old induction variables have to be replaced with new induction variables.

3.7 Algorithm choices and compilation speed

Polyhedral optimizers have challenges with scalability and GRAPHITE is no exception. While developing GRAPHITE, we have encountered some interesting issues that affect compilation speed and are exploring algorithms choices to improve performance.

One example is loop unrolling. C++ templates are a powerful feature used in many high-performance codes; template meta-programming is combined with inlining to produce specialized loops. This style creates a large abstraction penalty that GCC has chosen to address with an early inner loop unrolling pass. Applications (such as Tramp3D in the GCC test suite) show significant performance improvement through this technique. However, this also affects loop and data dependence analysis for optimizations such as auto-vectorization and GRAPHITE, whose analysis and compilation time grows with the number of variables in each SCoP. We explore ways to tune this unrolling in the presence of GRAPHITE and eventually to implement such unrolling within GRAPHITE itself.

4 Optimizations

Loops that carry no dependence may be good candidates to be parallelized, i.e, different iterations of the loop might be executed simultaneously by multiple threads [1]. In GCC, two main infrastructures are used to accomplish loop parallelization: data dependency analysis and the GNU OpenMP library. OpenMP defines language extensions to C, C++, and Fortran for implementing multithreaded shared memory applications [29]. Automatic generation of such extensions by the compiler relieves programmers from the manual parallelization process. OpenMP support has been implemented in GCC since version 4.2 [27], and together with existing data dependence analyses, opened the door for automatic parallelization in GCC.

Automatic parallelization was first implemented in version 4.3 as a technology preview. It is able to detect loops carrying no dependences, and generate parallel code by creating and inserting the necessary OpenMP structures and support. It is triggered by `-ftree-parallelize-loops=`x, where x defines the number of threads to create.

Generating parallel code Once the GCC auto-parallelizer decides to parallelize a loop, it generates the parallel code using the OpenMP structures to define the parallel section and relevant attributes like the scheduling method, the shared vs. private variables, atomic operations etc.

In Figure 9, we see an example of a sequential loop, and the parallel code generated for it (assuming the number of threads requested by the user is 4). `.paral_data` is a structure field gathering all the shared data that should be provided for each thread. The loop is outlined to a separate function, `parloop._loopfn()`, which is run individually by each thread, supplied with the shared data.

The GNU OpenMP library provides two builtins which define the parallel section: `GOMP_parallel_start()` creates the threads. `GOMP_parallel_end()` is a barrier where all the threads are joined. After `GOMP_parallel_start()` is executed, 4 threads are created. Each thread is executing the outlined function, iterating the loop with a different (and exclusive) interval of iterations, represented as start and end at the example. After `GOMP_parallel_end()` is executed, the threads are joined back to one master thread.

```
parloop
{
  for (i = 0; i < N; i++)
    x[i] = i + 3;
}
```

(a)

```
parloop
{
  .paral_data.x = &x;
  __builtin_GOMP_parallel_start (parloop._loopfn,
                                 &.paral_data, 4);
  parloop._loopfn (&.paral_data);

  __builtin_GOMP_parallel_end ();
}

parloop._loopfn (.paral_data)
{
  for (i = start; i < end; i++)
    (*.paral_data->x)[i] = i + 3;
}
```

(b)

Fig. 9. (a) sequential loop (b) parallel code generated

```
for i
  for j
    A[i][j] = A[i-1][j]
```

(a)

```
for j
  for i
    A[i][j] = A[i-1][j]
```

(b)

Fig. 10. (a) the original loop (b) after interchange

Integration of the parallelizer with graphite The initial analysis used for the parallelizer was based on the Lambda framework [24]. It has been replaced with the GRAPHITE based dependence analysis.

Integrating the parallelizer with GRAPHITE is profitable for a number of reasons:

- GRAPHITE dependence analysis is more accurate than Lambda, hence could detect more parallel loops [38].
- The ability of GRAPHITE to perform long and complex compositions of program transformations enables to extract more parallelism [13] and to optimize for parallelism and locality simultaneously [7].
- Since GRAPHITE is able to represent sequences of loop transformations as a single scheduling transformation, it seems natural to incorporate a cost model into it to control the transformation sequence. Parallelization is a key transformation whose cost and benefit should be applied to such a model, in the hope of deriving the

most profitable combination of loop transformations. We worked on such a cost model in the special case of automatic vectorization [37], extension to more general parallelization and to the management of temporal locality is in progress.

Figure 10 shows a simple example demonstrating the interaction of loop parallelization with another transformation, loop interchange. The original loop is shown in Figure 10(a). The outer loop carries a dependence and therefore can't be parallelized. Parallelizing the inner loop is possible, but results in executing a synchronization barrier at the end of each outer-loop iteration, therefore executing a synchronization `i` times. If, however, we interchange the loop, as shown in (b), we can parallelize the outer-loop, resulting in use of just one barrier.

Automatic parallelization was integrated to GRAPHITE as part of the upcoming GCC4.5. In addition to `-ftree-parallelize-loops=`x, `-floop-parallelize-all` is specified to enable it as a GRAPHITE-based transformation.

5 Alias Information and Polyhedra

Alias analysis is an intrinsic module of any compiler as it facilitates any other optimization that involves variable disambiguation, such as scheduling or identifying invariants, redundant subexpressions, etc. For scalability reasons, most compilers use fast but rather imprecise analysis like Anderson's algorithm [2], a context-insensitive, flow-insensitive subset-based may-alias analysis. GCC relies on an extension of this algorithm that is field-sensitive as well [5, 30].

A data-reference is either a scalar variable, or an array-reference, or an offset of an array by a compile-time constant, or an offset of an array by an index, or a pointer variable. The difference between the latter four types in `C` is that the first three resolve to a constant pointer (like `const int *`) referring to a stack location, while the last one can only be resolved to an ordinary pointer (like `int *`) referring to a heap location.

Example In the following code excerpt, it can be seen that a and p may-alias to each other, and so do p and b, but a and b do not.

```
int a[10], b[10];
void foo (int *p);
```

Most alias analysis algorithms return a points-to relation, where data references are mapped to abstract stack or heap locations called *alias sets*. We will also refer to this relation as the forward mapping. On the above example it is: $a \to \{A_1\}$, $p \to \{A_1, A_2\}$, $b \to \{A_2\}$.

In GCC (since version 4.4), the result of the alias-analysis is encoded in an *alias oracle* that returns the information about *presence or absence* of may-alias relation between pairs of data references. It can be seen that such a portable interface goes well with various scalar analyses that use it.

The information that is provided by alias-oracle can be represented as an undirected graph, whose vertices correspond to data-references and whose edges represent presence of alias-relationship between pairs of data-references.

The aliasing relation for the previous example can be represented as a graph G_a: a — p — b.

It is known that polyhedral dependence analysis for a given SCoP usually makes $\mathcal{O}(n^2)$ polyhedral operations, when n is the number of convex polyhedra representing memory references. Dependence analysis can exploit the properties of G_a such that the number of calls to polyhedral libraries can be reduced. The above examples show that the dependence analysis could effectively use the information provided by the alias-analysis to its full potential.

To represent the alias information, GRAPHITE creates an additional dimension (the first) for each array reference. This additional dimension, henceforth called *alias dimension* is indexed by the alias set to which that particular data reference points to. A data reference however, could be a member of more than one alias set. For example, variable p in the above example, belongs to two alias-sets A_1 and A_2. Though this may be because of impreciseness of the algorithm used in the alias analysis, it could as well be because of a true aliasing of the associated memory regions. Hence, GRAPHITE indexes the alias dimension by the disjunction of alias-sets to which that particular data-reference points to.

In the above example, if we let $\mathcal{D}_a^R$, $\mathcal{D}_p^R$ and $\mathcal{D}_b^R$ be the original polyhedral domains of the three variables a, p and b, then the corresponding memory references, annotated by the alias dimension would be $\texttt{memory\_reference}[A_1, \mathcal{D}_a^R]$, $\texttt{memory\_reference}[A_1 \vee A_2, \mathcal{D}_p^R]$, and $\texttt{memory\_reference}[A_2, \mathcal{D}_b^R]$ respectively.

Definition: Minimum Edge Clique Cover For an undirected graph $G = (V, E)$, the Minimum-Edge Clique Cover is defined to be a collection $A_1, A_2, \ldots, A_k$ of subsets of V, such that each A_i induces a complete subgraph of G and such that for each edge $(u, v) \in E$, there is some A_i that contains both u and v. This problem is also called *Edge Clique Cover* (ECC). Another problem similar to ECC is the *Vertex Clique Cover* (VCC) that computes a cover on cliques of vertices.

It is easy to see that if there is a clique in G_a, then polyhedral dependence analysis should test for dependence between all variables participating in the clique to determine the nature of dependence between the data-references participating in the clique. If on the other hand, there is no edge between a pair of variables, there is no need for polyhedral operations. This information is equivalent to *cliques* in G_a. In the above example, it can be seen that there are two 2-cliques: $\{a, p\}$ and $\{p, b\}$. Polyhedral dependence analysis should test for dependence between each of these pairs. It however does not need to test for dependence between a and b.

It is clear that maximizing the clique size in G_a is helpful. But, it is the edges of G_a that correspond to possible intersections of memory areas, thereby corresponding to aliasing of data-references. Hence, the problem that minimizes the number of representative elements in the alias-dimension should be an edge-clique, thereby meaning ECC (rather than vertex-clique or maximum clique). Further, as the edges are *covered* by the representative element, along with all possible edges which could alias with it, the dependence analysis can thus use alias-set representatives to drive its algorithm.

5.1 ECC: problem and solution

The ECC problem is an NP-Hard problem (page 194 in [12]). It is different from the more widely studied VCC problem, which is closely related to the graph coloring problem. The VCC solution for a graph G, though being a NP-Hard problem itself – could be trivially found after coloring the complement graph G'.

No fast running and close to optimal heuristic is known for ECC. It may be very hard to solve even in an approximate sense. On the other hand, the other above mentioned problems (VCC and graph coloring) have linear-time and exact solutions for special classes [14].

The algorithms for ECC either solve exactly by a brute-force search, or by using a heuristic developed for VCC or graph-coloring. In Gramm et. al's paper [15], which is state of the art for this problem, new algorithms for both methods are suggested. In their paper, the running time of a previously known heuristic is improved from $\mathcal{O}(|V||E|^2)$ to a more acceptable $\mathcal{O}(|V||E|)$. The major contribution of the paper however, are data-reduction rules that help reduce the input problem size by preprocessing. The method suggested in [15] is that the resultant graph after iteratively applying the rules is usually smaller, and hence can be subjected to either of the above mentioned methods (exact-solution or heuristic) for a faster solution.

5.2 Empirical analysis of alias graphs

We have done an empirical analysis of the graphs that are returned by the alias-analysis currently in GCC.

Of the 4481 graphs from SPEC Cpu 2006 benchmarks, 4367 are trivial, with the definition of trivial being $|V| < 10 \vee |E| < 5$. Only 114 graphs are non-trivial. Of the latter kind of graphs, only 11 graphs are interesting. In the rest of the graphs, every connected component is a clique. In all the graphs, the number of vertices participating in the maximal cliques vary in the range $1 \leq |V| \leq 90$.

From the above empirical analysis, one could say that the alias-oracle which marks every connected component as a clique is generally very imprecise and hence advanced algorithms for solving ECC are not needed in the present context. A general counter-example to such a reasoning is the large size of graphs, taken from real-world examples, containing a wide range of maximal cliques. Further, we also have specific counter-examples which show that exceptions to the above statement exist in real world. Figure 11 depicts the alias relation for a kernel in the GCC-testsuite extracted from an H.264 decoder. Each ellipse on the (i) graph represents a clique. A block-edge between two ellipses X and Y represents an edge between every pair of vertices in the set $\{X, Y\}$.

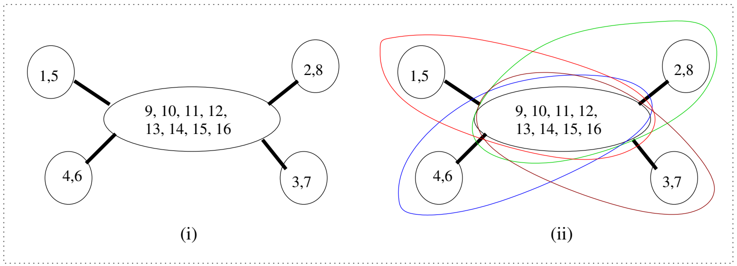


Fig. 11. Alias relation for an H.264 kernel.

The optimal solution of ECC is shown on the (ii) graph. It has 4 cliques, each of which is a union of one of the smaller cliques with the biggest clique.

We now describe the polynomial algorithm that we chose to compute an ECC:

Input: an alias graph G_a

Output: a mapping from vertices to the alias-set representatives.

- Do a Depth First Search (DFS) on G_a and separate out the connected components $A_1, A_2, \ldots, A_k$. This step takes $\max(|V(G_a)|, |E(G_a)|)$ time.
- Check if each connected component A_i is a clique or not. This step takes $|V(A_i)|^2$ time, where $V(A_i)$ is the number of nodes of the connected component A_i.

- If A_i is small enough ($|V| < 5 \vee |E| < 10$), then search for solution using a direct search.
- Apply the simple and cheaper data-reduction rules (mainly Rule 1 and Rule 2), as explained in [15], so that the problem size could be reduced.

Currently, we are using DFS based numbering, testing for cliques, and simple search. Though this method is very conservative, it gives the optimal solution for most of the cases for SPEC Cpu 2006, though not for the H.264 example shown above. For this example, this solution returns that all edges are in the same cover. Thus our current method is not searching for a clique, which leads to loss of precision, and needs to be improved. We are working on another heuristic approach to design a polyhedral algorithm computing a suboptimal ECC but without loss of points-to information.

6 Conclusion

We presented the design of the GRAPHITE pass of GCC, focusing on the challenges and novel research issues arising from this confrontation of polyhedral compilation with the real world. Our work makes the following contributions:

- We implemented the polyhedral model on a three-address, SSA-based representation, opening interesting reuse and interaction opportunities for analyses and optimizations in production compilers.
- We extended the polyhedral representation to capture alias relations among pointer-based data references, with no impact on polyhedral dependence analysis and transformation algorithms.
- We also extended this representation to capture scalar dependences and reductions.
- We set the framework for aggregating statements into “polyhedral basic blocks” or splitting those blocks into smaller components, with the ability to trade expressiveness for compilation time.
- We motivated further research on the practical interaction between polyhedral loop transformations and other optimizations, including parallelization and vectorization.

6.1 Prospective work

There are two main issues that are the focus of the prospective work on automatic parallelization:

- **Heuristics/cost model for automatic parallelization**. Currently, a very simple method is used to determine whether it is profitable to parallelize a certain loop. We need a good model to determine if we should parallelize a loop considering performance reasons.
- **Advanced automatic parallelization**. Currently we are only able to detect whether a loop is parallel or not. We would like to explicitly apply transformations to increase and expose further parallelism opportunities, and we have shown that GRAPHITE is the right setting to design such transformations.

Acknowledgments. Tobias Grosser was supported by AMD as a summer intern and by a Google Summer of Code grant. Konrad Trifunovic was supported by IBM and the HiPEAC FP7 European network as a summer intern. Li Feng was supported by a Google Summer of Code grant. GRAPHITE was partially supported by the ACOTES FP6 European project.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan and Kaufman, 2002.
2. L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, 1994.
3. 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.
4. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, Sept. 2004.
5. D. Berlin. Structure aliasing in GCC. In *the GCC Developers' Summit*, pages 25–36, 2005. http://www.gccsummit.org/2005.
6. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.
7. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'08)*, Tucson, AZ, USA, June 2008.
8. C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.
9. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.
10. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.
11. 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.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
13. 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.
14. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Elsevier, 2nd edition, 2004.
15. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction and exact algorithms for clique cover. *J. Exp. Algorithmics*, 13:2.2–2.15, 2009.
16. M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
17. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, Cologne, Germany, June 1991.
18. R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, New York, NY, USA, 1994. ACM.
19. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. http://www.hp.com/techsevers/software/kap.html.
20. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

21. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.
22. W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, 1993.
23. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.
24. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.
25. The LooPo Project - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.
26. B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for pgas platforms with the r-stream compiler. In *APGAS'09 Workshop on Asynchrony in the PGAS Programming Model*, Yorktown Heights, New York, June 2009.
27. D. Novillo. Openmp and automatic parallelization in gcc. In *the GCC Developer's summit*, June 2006.
28. M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.
29. The OpenMP API specification for parallel programming. http://openmp.org/wp/.
30. D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.
31. PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.
32. S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4^{th} GCC Developper's Summit*, Ottawa, Canada, June 2006.
33. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, Nov. 2005. Springer-Verlag.
34. L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM Conf. on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 2008.
35. L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.
36. A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.
37. 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.
38. N. Vasilache, A. Cohen, C. Bastoul, and S. Girbal. Violated dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'06)*, Cairns, Australia, June 2006.
39. N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, Romania, Sept. 2007.