GRAPHITE: Polyhedral Analyses and Optimizations for GCC

Sebastian Pop [1] Albert Cohen [2] Cédric Bastoul [2] Sylvain Girbal [2]
Georges-André Silber [1]
Nicolas Vasilache [2]

[1] *CRI, École des mines de Paris, Fontainebleau, France*
`lastname@cri.ensmp.fr`
[2] *Alchemy group, INRIA Futurs and LRI, Paris-Sud 11 University, Orsay, France*
`firstname.lastname@inria.fr`

Abstract

We present a plan to add loop nest optimizations in GCC based on polyhedral representations of loop nests. We advocate a static analysis approach based on a hierarchy of interchangeable abstractions with solvers that range from the exact solvers such as OMEGA, to faster but less precise solvers based on more coarse abstractions. The intermediate representation GRAPHITE[1] (GIMPLE Represented as Polyhedra with Interchangeable Envelopes), built on GIMPLE and the natural loops, hosts the high level loop transformations. We base this presentation on the WRaP-IT project developed in the Alchemy group at INRIA Futurs and Paris-Sud University, on the PIPS compiler developed at École des mines de Paris, and on a joint work with several members of the static analysis and polyhedral compilation community in France.

The main goal of this project is to bring more high level loop optimizations to GCC: loop fusion, tiling, strip mining, etc. Thanks to the WRaP-IT experience, we know that the polyhedral analyzes and transformations are affordable in a production compiler. A second goal of this project is to experiment with compile time reduction versus attainable precision when replacing operations on polyhedra with faster operations on more abstract domains. However, the use of a too coarse representation for computing might also result in an over approximated solution that cannot be used in subsequent computations. There exists a trade off between speed of the computation and the attainable precision that has not yet been analyzed for real world programs.

1 Introduction

Static compiler optimizations can hardly cope with the complex run-time behavior and hardware components interplay of modern processor architectures. Multiple architectural phenomena occur and interact simultaneously. The optimizer needs to combine multiple program transformations to harness the computing and

[1] This work was partially supported by ACI/APRON.

storage resources and to fight all sources of pipeline stalls or flushes. In addition, conventional processor architectures are shifting towards coarser grain on-chip parallelism, to avoid diminishing returns of further extending instruction-level parallelism. This shift rejuvenates the hard static analysis and optimization problems associated with automatic parallelization (extraction, exploitation and optimization of parallelism).

Even provided with enough static information or annotations (OpenMP directives, pointer aliasing, separate compilation assumptions), compilers have a hard time exploring the huge and unstructured search space associated with these application-to-architecture mapping and optimization challenges [16, 32, 23, 53, 1]. In a sense, the task of the compiler can hardly be called optimization anymore, in the traditional meaning of lowering the abstraction penalty of a higher-level language. Together with the run-time system (whether implemented in software or hardware), the compiler is responsible for most of the combinatorial code generation decisions to map the simplified and idealistic operational semantics of the source program to the highly complex and heterogeneous machine.

Unfortunately, optimizing compilers have traditionally been limited to systematic and tedious tasks that are either not accessible to the programmer (e.g., instruction selection, register allocation) or that the programmer in a high level language does not want to deal with (e.g., constant propagation, partial redundancy elimination, dead-code elimination, control-flow optimizations). Generating efficient code for deep parallelism and deep memory hierarchies with complex and dynamic hardware components is a completely different story: the compiler (and run-time system) now has to take the burden of much smarter tasks that only expert programmers would be able to carry.

Recent work showed that polyhedral compilation techniques are good candidates to address these challenges, and that new algorithms allow them to scale to real-size optimization problems (beyond tiny loop kernels) [34, 64]. This paper exposes our road-map towards making GCC the first general-purpose compiler to build on full-scale polyhedral compilation techniques (analysis and transformations, including affine scheduling).

In the first part of the paper we will present the steps to transform the loops and GIMPLE representations to systems of linear constraints, or polyhedra, to transform the matrix form obtained, and to eventually regenerate GIMPLE trees. This part corresponds to an adaptation to GCC of the WRaP-IT tool. Then, we discuss the integration of additional numerical domains, to support a wider range of (interprocedural) static analyses, and to improve the compile time on polyhedral compilation passes. This work will use the APRON library as a starting point, to facilitate the transparent substitution of abstract numerical domains. The APRON library is part of a joint work between different members of the static analysis community in France, and aims at providing a common interface between numerical abstract domains. Because some computations might not be tractable, or too expensive on a too precise representation, it is interesting to use more abstract representations on which computations have lower costs.

We will present experimental results to motivate the polyhedral program transformation approach, and we will survey our methods to let the code analysis and generation techniques scale to full-scale loop nests (with aggressive inlining). We will present the benefits of adopting this infrastructure: composition of transformations, and interchangeability of abstract domains.

2 State of the art

In compilers, polyhedral domains are used for different purposes:

Static analysis. Polyhedra represent conservative approximations of the properties of a program [24]. In this case, the operations on the abstract domain should preserve the safety of the computed properties, and thus the results are allowed to be over approximations.

Code transformations. Polyhedra represent the code itself [31], through the iteration domain, iteration and statement schedules, and memory access functions. The translation and the operations over the polyhedral representation have to be exact (with no loss of information), to guarantee that the code generated from the polyhedral representation after transformation will be semantically equivalent to the original program.

Several works addressed these applications in different experimental frameworks, but the underlying mathematical framework is the same.

In this section we provide an overview of the techniques used in research and industrial compilers based on polyhedral domains: first we present the polyhedral representations for loop iteration domains, then we present the array regions that approximate data accesses. We end this survey with the cost models based on the polyhedral representations.

2.1 Translation to a Polyhedral Representation

The polyhedral representations are restricted by their expressiveness to represent only sequences of loop nests with constant strides and affine bounds. It includes non-rectangular loops, non-perfectly nested loops, and conditionals with boolean expressions of affine inequalities. Loop nests fulfilling these hypotheses are amenable to a representation in the polyhedral model [54]. We call *Static Control Part* (SCoP) any maximal syntactic program segment satisfying these constraints [20]. The reader interested in techniques to extend SCoP coverage (by preliminary transformations) or in partial solutions on how to remove this scoping limitation (procedure abstractions, irregular control structures, etc.) should refer to [62, 37, 21, 72, 25, 12, 60, 11, 19, 22].

In the polyhedral model [57, 31], the iteration steps of a loop nest of depth d are represented as the integer points of a polyhedra in $\mathbb{Z}^d$. In the general case, the polyhedra are bounded by symbolic parameters: they are called parametric polyhedra, and each symbolic parameter is represented using an extra dimension. All variables that are invariant within a SCoP are called *global parameters*. For each statement within a SCoP, the representation separates four attributes, characterized by parameter matrices: the iteration domain, the schedule, the data layout and the access functions.

2.2 WRaP-IT

The WRaP-IT framework [34], developed in the Alchemy group, improves on classical polyhedral representations [31, 68, 41, 46, 2, 47] to support a large array of useful and efficient program transformations (loop fusion, tiling, array forward substitution, statement reordering, software pipelining, array padding, etc.), as well as *compositions* of these transformations. It is implemented within the Open64 and PathScale EKOPath [17] compilers. This compiler family provides key interprocedural analyses and pre-optimization phases such as inlining, interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, as well as induction variable substitution. Thanks to these preliminary passes, our tool extracts large and representative SCoP for SPEC fp benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop. See [34] for detailed static and dynamic SCoP coverage. GCC now comes close to Open64 in terms of loop-oriented program normalizations, and the situation improves quickly; our future research will thus benefit from migrating the WRaP-IT framework to GCC.

The main technical idea behind polyhedral program representations is to clearly separate the four different types of actions performed by loop-centric transformations: modification of the iteration domain (loop bounds and strides), modification of the schedule of each individual statement, modification of the access functions (array subscripts), and modification of the data layout (array declarations). This separation makes it possible to provide a matrix representation for each kind of action, enabling the easy and independent composition of the different representation operations associated with each program transformation, and as a result, enabling the composition of transformations themselves. Current representations do not clearly separate these four types of actions; as a result, the implementation of certain compositions of program transformations can be complicated or even impossible. For instance, current implementations of loop fusion must include loop bounds and array subscript modifications even though they are only byproducts of a schedule-oriented program transformation; after applying loop fusion, target loops are often peeled, increasing code size and making further optimizations more complex. Within our representation, loop fusion is only expressed as a schedule transformation, and the modifications of the iteration domain and access functions are implicitly handled, so that the code complexity is exactly the same before and after fusion. Similarly, an iteration domain-oriented transformation like unrolling should have no impact on the schedule or data layout representations; or a data layout-oriented transformation like padding should have no impact on the schedule or iteration domain representations.

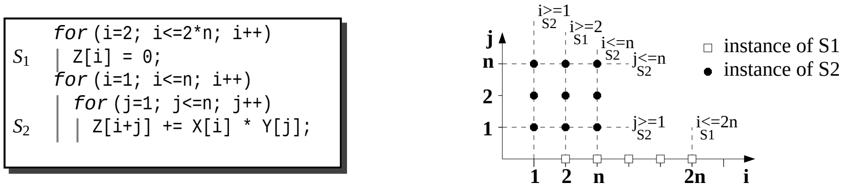
3 Loop Transformations in the Polyhedral Model

This section is a quick overview of the polyhedral framework and shows the expressiveness benefits on a practical example. A more formal presentation of the model may be found in [57, 31].

3.1 Quick Overview of the Framework

Polyhedral compilation usually distinguishes three steps: one first has to represent an input program in the formalism, then apply a transformation to this representation, and finally generate the target (syntactic) code.

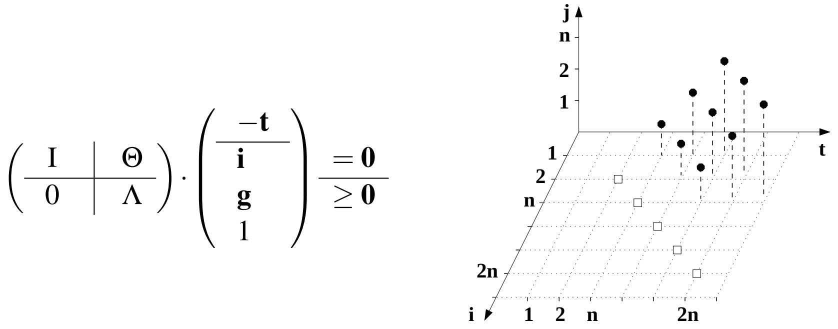
Consider the polynomial multiplication kernel in Figure 1(a). It only deals with control aspects of the program, and we refer to the two computational statements (array assignments) through their names, S_1 and S_2. To bypass the limitations of syntactic representations, the polyhedral model is closer to the execution itself by considering *statement instances*. For each statement we consider the *iteration domain*, where every statement instance belongs. The domains are described using affine constraints that can be extracted from the program control. For example, the iteration domain of statement S_1, called D^{S_1}, is the set of values (i)



```
     for (i=2; i<=2*n; i++)
S1   | Z[i] = 0;
     for (i=1; i<=n; i++)
     | for (j=1; j<=n; j++)
S2   | | Z[i+j] += X[i] * Y[j];
```

(a) Syntactic form (b) Polyhedral domains ($n \geq 2$)

Figure 1: A polynomial multiplication kernel and its polyhedral domains



$$\left(\begin{array}{c|c} \mathrm{I} & \Theta \\ \hline 0 & \Lambda \end{array}\right) \cdot \left(\begin{array}{c} -\mathbf{t} \\ \hline \mathbf{i} \\ \mathbf{g} \\ 1 \end{array}\right) \begin{array}{c} = \mathbf{0} \\ \hline \geq \mathbf{0} \end{array}$$

(a) Transformation template formula (b) Transformed polyhedra

Figure 2: Transformation template and its application

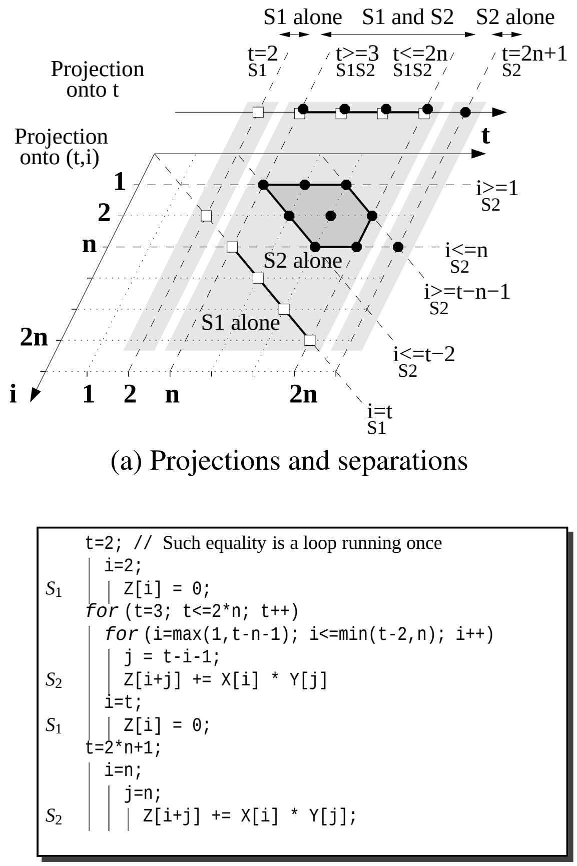
such that $2 \leq i \leq n$ as shown in Figure 1(b); a matrix representation is used to represent such constraints: in our example, D^{S_1} is characterized by

$$\left[\begin{array}{rrr} 1 & 0 & -2 \\ -1 & 2 & 0 \end{array}\right] \left(\begin{array}{c} i \\ n \\ 1 \end{array}\right) \geq \mathbf{0}.$$

In this framework, a transformation of the execution order is characterized by *affine scheduling functions* Θ^S, for all statements S in the SCoP. Each statement has its own scheduling function which maps each run-time statement instance to a logical execution date. In our polynomial multiplication example, an optimizer may notice a locality problem and discover a good data reuse potential over array Z, then suggest $\Theta^{S_1}(i) = (i)$ and $\Theta^{S_2}\left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) = (i + j + 1)$ to achieve better locality (see e.g., [14] for a method to compute such functions). The intuition behind such transformation is to execute consecutively the instances of S_2 having the same $i+j$ value (thus accessing the same array element of Z) and to ensure that the initialization of each element is executed by S_1 just before the first instance of S_2 referring this element. In the polyhedral model, a transformation is applied following the template formula in Figure 2(a) [13], where $\mathbf{i}$ is the iteration vector, $\mathbf{g}$ is the vector of constant parameters, and $\mathbf{t}$ is the *time-vector*, i.e. the vector of the scheduling dimensions. The nature of these vectors and the structure of the Θ and Λ matrices is detailed in [34]. Notice that in this formula, equality constraints capture schedule modifications, and inequality constraints capture iteration domain modifications. The resulting polyhedra for our example are shown in Figure 2(b), with the additional dimension t.

Once transformations have been applied in the polyhedral model, one needs to (re)generate the target code. The best syntax tree construction scheme consists in a recursive application of domain projections and separations [59, 13]. The final code is deduced from the set of constraints describing the polyhedra attached to each node in the tree. In our example, the first step is a projection onto the first dimension t, followed by a separation into disjoint polyhedra, as shown on the top of Figure 3(a). This builds the outer loops of the target code (the loops with iterator t in Figure 3(b)). The same process is applied onto the first two dimensions (bottom of Figure 3(a)) to build the second loop level and so on. The final code is shown in Figure 3(b) (the reader may care to verify that this solution maximally exploits temporal reuse of array Z). Note that the separation step for two polyhedra needs three operations: $D^{S_1} - D^{S_2}$, $D^{S_2} - D^{S_1}$ and $D^{S_2} \cap D^{S_1}$, thus for n statements the worst-case complexity is 3^n.

It is interesting to note that the target code, although obtained after only one transformation step, is quite different from the original loop nest. Indeed, multiple classical loop transformations are necessary to simulate this one-step optimization (among them, software pipelining and skewing). The intuition is that arbitrarily complex compositions of classical transformations can be captured in one single transformation step of the polyhedral model. This was best illustrated by affine scheduling [31, 41] and partitioning [46] algorithms. Yet, because black-box, model-based optimizers fail on modern processors, we propose to step back a little bit *and consider again the benefits of composing classical loop transformations, but using a polyhedral representation.* Indeed, before our recent work, polyhedral optimization frameworks have only considered the isolated application of one arbitrarily complex affine transformation. The main originality of our work is to address the *composition of program transformations on the polyhedral representation itself*, which vastly facilitates the coordination of polyhedral transformations with classical heuristics and cost models, enabling their integration into production compilers.



(a) Projections and separations

```
     t=2; // Such equality is a loop running once
     | i=2;
S1   | | Z[i] = 0;
     for (t=3; t<=2*n; t++)
     | for (i=max(1,t-n-1); i<=min(t-2,n); i++)
     | | j = t-i-1;
S2   | | Z[i+j] += X[i] * Y[j]
     | i=t;
S1   | | Z[i] = 0;
     t=2*n+1;
     | i=n;
     | | j=n;
S2   | | | Z[i+j] += X[i] * Y[j];
```

(b) Target code

Figure 3: Target code generation

3.2 Code Generation from the Polyhedral Model

Regenerating syntax trees from affine schedules is one of the most time-consuming parts of the polyhedral compilation flow. The history of code generation in the polyhedral model shows a constant growth in transformation complexity, from basic schedules for a single statement to general affine transformations for wide code regions. In their seminal work, Ancourt and Irigoin limited transformations to unimodular functions (determinant 1 or -1) and the code generation process was applicable for only one domain at once [5]. Several works succeeded in relaxing the unimodularity constraint to invertibility (the T matrix has to be invertible), enlarging the set of possible transformations [27, 45]. A further step has been achieved by Kelly et al. by considering more than one domain and multiple scheduling functions at the same time [42]. All these methods relied on the Fourier-Motzkin elimination method [61] to build the target code.

Quilleré et al. showed how to use polyhedral operations based on the Chernikova Algorithm [66] instead, to benefit from its practical efficiency to handle bigger problems [59]. Recently, a new transformation policy has been proposed to allow general non-invertible, non-uniform, non-integral affine transformations [13, 64]. Such freedom allowed to apply polyhedral techniques to much larger programs with very sophisticated transformations, and led to novel complexity, scalability and code quality challenges we discuss in this paper. In the context of GCC, it would be very interesting to try to preserve the robustness of the Quilleré algorithm, but further improve the complexity of the method, using depth-sensitive relaxations (approximations) and Fourier-Motzkin eliminations.

3.3 Optimization Experiment

We applied the WRaP-IT tool to the swim SPEC CPU2000 fp benchmark, extracting several SCoP: aggressive inlining yields one SCoP of 421 lines of code — 112 instructions in the polyhedral representation — in consecutive loop nests within the `main` function. We applied more than 30 transformations to this SCoP, including multi-level loop fusion, loop shifting (pipelining), loop tiling, loop peeling, loop unrolling, loop interchange, and strip-mining [71, 4]. All these transformations are generalized to non-perfectly nested codes, and embedded in our compositional framework [34].

The resulting code is significantly larger — 2267 lines — roughly one third of them being naive scalar copies to map schedule iterators to domain ones, fully eliminated by copy-propagation in the subsequent run of EKOPath or Open64. This is not surprising since most transformations in the script require domain decomposition, either explicitly (peeling) or implicitly (shifting prolog/epilog, at code generation). It takes 39s to apply the whole transformation sequence up to native code generation on a 2.08GHz AthlonXP. Transformation time is dominated by back-end compilation (22s). Polyhedral code generation takes only 4s. Exact polyhedral dependence analysis (computation and checking) is acceptable (12s). Applying the transformation sequence itself is negligible. These execution times are very encouraging, given the complex overlap of peeled polyhedra in the code generation phase, and since the full dependence graph captures the exact dependence information for the 215 array references in the SCoP at every loop depth (maximum 5 after tiling), yielding a total of 441 dependence matrices.

Compared to the *peak performance attainable by the best available compiler*, PathScale EKOPath (V2.1) with the *peak-SPEC* optimization flags, our tool achieves **32% speedup on Athlon XP and 38% speedup on Athlon 64**. Compared to the *base-SPEC* performance numbers, our optimization achieves **51% speedup on Athlon XP and 92% speedup on Athlon 64**. We are not aware of any other optimization effort — manual or automatic — that brought swim to this level of performance on $x86$ processors.[2]

[2] Notice we consider the SPEC CPU2000 version of swim, much harder to optimize through loop fusion than the SPEC 95 version.

4 Static Analysis with Polyhedra

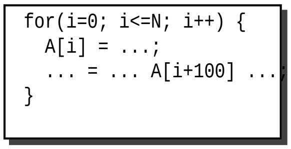
Let us now discuss some of the static analysis opportunities and challenges offered by polyhedral methods.

4.1 Instancewise Polyhedral Dependence Analysis

Many tests have been designed for dependence checking between different statements or between different executions of the same statement. It has been shown that this problem is equivalent to detecting whether a system of equations has an integer solution inside a region of $\mathbb{Z}^n$ [9].

Most of the dependence tests try to find efficiently a reliable, approximative but conservative (they overestimate data dependences) solutions. The GCD-test [8] has been the very first practical solution, it is still present in many implementations as a first check with low computational cost. This test assumes that if the greatest common divisor of the coefficients of an equation divides the constant term, then a solution exists. A generalized GCD-test has been proposed to handle multi-dimensional array references [9]. The Banerjee test uses the intermediate value theorem to disprove a dependence: it computes the upper and lower bounds of an equation and checks if the constant part lies in that range [69]. The λ-test is an extension to this test that handles multi-dimensional array references [43]. Some other important solutions are a combination of GCD and Banerjee tests called I-test [43], the Δ-test [35] that gives an exact solution when there is at most one variable in the subscript functions, and the Power-test which uses the Fourier-Motzkin variable elimination method [61] to prove or disprove dependences [70]. Beside their approximative nature, these dependence tests suffer from many other major limitations. The most stringent one is their inability to precisely handle `if` conditionals, loops with parametric bounds, triangular loops (a loop bound depends on an outer loop counter), coupled subscripts (two different array subscripts refer the same loop counter), or parametric subscripts.

On the opposite, a few methods allow to find an exact solution to the dependence problem, but at a higher computational cost. The OMEGA-test is an extension to the Fourier-Motzkin variable elimination method to find integral solutions [55]. On one hand, once a variable is eliminated, the original system has an integer solution only if the new system has an integer solution (if this is not the case there is no solution). On the other hand, if an integer point exists in a space computed from the new system, then there exists an integer point in the original system (if this is the case, there is a solution). The PIP-test uses a parametric version of the dual-simplex method with Gomory cuts to find an integral solution [30]. These two tests not only give an exact answer, they are also able to deal with complex loop structures and (affine) array subscripts. The PIP-test is more precise than the OMEGA-test when dealing with parametric codes (when one or more integer symbolic constant are present), for instance, in the following pseudo-code:



```
for(i=0; i<=N; i++) {
  A[i] = ...;
  ... = ... A[i+100] ...;
}
```

the OMEGA-test will state that there is a dependence between the two statements while the PIP-test will precise that the dependence only exists if N is greater or equal to 100. Both tests have worst-case exponential complexities but work quite well in practice as shown by Pugh for the OMEGA-test [55]. Other costly exact tests exist in the literature [49, 28] but are often not able to handle complex control in spite of their cost.

We do not advocate for the use of any of these tests, but rather for the computation of *instancewise* dependence information as precisely as possible, i.e., for intensionally describing the statically unbounded set of all pairs of dependent statement iterations, called *instances*. Dependence tests are statementwise decision problems associated with the existence of a pair of dependent instances, while instancewise dependence analysis provides additional information that can enable finer program transformations, like affine scheduling [44, 31, 46, 36]. The intensional characterization of instancewise dependences can take the form of multiple *dependence abstractions*, depending on the precision of the analysis and on the requirements of the user. The simplest and least precise one is called *dependence levels*, it specifies for a given loop nest which loop carry the dependence. It has been introduced in the Allen and Kennedy parallelization algorithm [3]. The *direction vectors* is a more precise abstraction where the *i*-th element approximates the value of all the *i*-th elements of the *distance vectors* (which shows the difference of the loop counters of two dependent instances). It has been introduced by Lamport [44] and formalized by Wolfe [71] and is clearly the most widely used representation. A more precise abstraction is the *dependence polyhedron* [40] which is able to determine exactly the set of statement instances in dependence relation. The choice of a given dependence abstraction is crucial for further study: choosing an imprecise one can result in blacking out interesting transformations. For instance, let us consider the following example:

```
 for(i=0; i<=N; i++)
   for(i=0; i<=N; i++)
S    A[i][j] = A[j][i] + A[i][j-1];
```

there are three dependences in this loop nest (a read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,i,j+1\rangle$, another read-after-write dependence from $\langle S,i,j\rangle$ to $\langle S,j,i\rangle$ and a write-after-read from $\langle S,j,i\rangle$ to $\langle S,i,j\rangle$). Dependence levels are 2, 1 and 1: each loop carries at least one dependence and no parallelism can be found. Direction vectors are $(0,1)$, $(+,-)$, $(+,-)$: the second coefficients 1 and $-$ hamper any parallelism detection. Using dependence polyhedra, parallelism may be found: the Feautrier algorithm suggests the affine schedule $\theta(i,j) = 2i+j-3$ (all instances with the same schedule may be run in parallel), see [67]. We propose to compute one of the most precise representation of dependences: the dependence polyhedra.

We exercise this implementation on 6 full SPEC CPU2000 fp benchmarks. In the most challenging examples, the biggest SCoP almost contains the whole program after inlining. On a 2.4GHz Pentium 4, the full instancewise dependence analysis takes up to 37.512 seconds, for the largest SCoP in applu. This is an extreme case with huge iteration spaces (more than 13 dimensions on average, and up to 19 dimensions). This may sound quite costly, but it still shows that the analysis is compatible with the typical execution time of aggressive optimizers (typically more than ten seconds for Open64 with interprocedural optimization and aggressive inlining and loop-nest optimizations). In all other cases, it takes **less than 5 seconds**, despite thousands of operations on polyhedra with close to 10 dimensions on average. These are very compelling results since we compute very large dependence graphs, taking *all pairs of references* into account, without *k*-limiting heuristic on their syntactic or nesting distance as it is the case in classical optimization frameworks. Also, a typical loop optimizer will perform on-demand computations on part of the dependence graph only.

4.2 Array Regions

In the context of interprocedural analysis of data dependences, array regions techniques have been proposed to extend the data dependence analysis. This representation is able to accurately describe sets of reads or writes that occur during the execution of a procedure. Approximations of accessed regions is not useful in general, as the precision degradation harm to the extraction of precise use-def chains, or dependence tests. As a practical implementation, the PIPS compiler [39, 38] uses the notion of transformers, or transfer functions for defining polyhedral relations between memory stores. Transformers are computed interprocedurally bottom-up accumulating the effects of procedures on memory accesses [26, 25], while preconditions are computed top-down accumulating a safe description of the state of the program when entering a procedure. Once this information has been gathered, the dependence test can be refined using the interprocedural information. A more advanced dependence test can be implemented by gathering more precise information: the in and out regions. The regions that contain both reads and writes potentially prevent the parallelization of a loop. When the data is written once and then read, the region can be selected to be privatized [63]. Because the duplication of runtime data can be harmful to the execution speed, the decision to privatize the data is deferred to an analyzer that can determine the benefit of the transformation.

4.3 Cost Models

Most of the loop nest transformations require a profitability analysis: often, several transformation schemes are available and following the specificities of the target architecture some strategies are preferable. The metrics developed for the polyhedral model are generally based on counting the number of integer points in polyhedra [58, 18, 65]. As polyhedra represent iteration spaces or data accesses, the evaluation of the number of integer points corresponds to the evaluation of the number of iterations of a loop nest, or the size or frequency of the accessed data. From this measure it is possible to infer other useful informations such as the memory bandwidth, cache reuse for the execution of a loop [48], cache misses [33], etc. Most of these methods are exponential, but some recent works [65] implemented and experimented with promising polynomial time algorithms for counting integer points in polyhedra.

5 Road Map

We describe the components that compose GRAPHITE, the priorities and dependences between the modules, and discuss the proposed plan for the integration into GCC. An overview of the modules is depicted in Figure 4: the development of the modules contains five stages. First the translation from GIMPLE to the polyhedral representation, then the translation back to GIMPLE, the development of cost models and the selection of the transform schedule. The interprocedural refinement of the data dependence information based on the array regions is optional, but it is necessary for gathering more precise informations that potentially could enable more transformations, or more precise transform decisions. Finally, the least critical component is the integration of the numerical domains common interface, based on which it will be possible to change the complexity of the algorithms used in the polyhedral analyses.

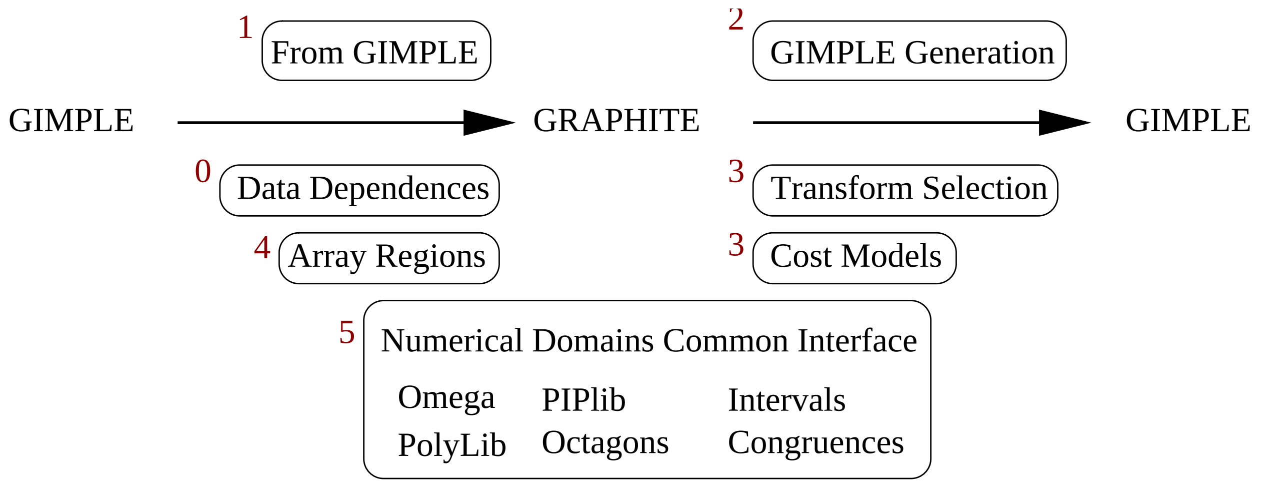


Figure 4: Overview of the modules composing GRAPHITE. The numbers indicate the order in which each module will be integrated.

5.1 Polyhedral Data Dependences

The code for statementwise dependence testing has been integrated in GCC and directly uses the OMEGA data structures for representing polyhedra.

Instancewise dependence analysis will be imported from the WRaP-IT framework, and extended to non-affine constructs and `while` loops [72, 10].

The main challenge here is memory usage, and the associated adaptation of dependence abstraction accuracy depending on some form of distance in the SCoP.

5.2 Translation to and from GRAPHITE

The next step for the integration of the GRAPHITE project is the translation of the code to the polyhedral representation and back to GIMPLE. This translation will be developed as an extension of the existing LAMBDA kernel [45, 15]. The main limitation of LAMBDA is that it works only on a single loop nest and on distance vectors, whereas more exact dependence informations like the polyhedral representation of the dependences among several sibling loop nests will be needed for GRAPHITE.

To support the code generation engine of GRAPHITE, we will only need a reduced set of operations on polyhedra [13, 64] — projection, difference and intersection — but these operations are very expensive when naively implemented on top of OMEGA. For this reason we will use a more efficient algorithm that is part of the PIP library [29].

After this basic infrastructure has been implemented, it is possible to transform the code by selecting the transformations either by hand, or let an expert system select the transformation sequence based on some metric.

5.3 Optimization Heuristics

Cost models and optimization selection could be implemented using different techniques. Purely static methods are based on the evaluation of statically computable properties, and are frequent in the classic heuristics for loop transformations. New heuristic techniques include cost models based on performance measurements either on abstract interpreters, simulators, or real hardware. The integration of this information in the compiler can be based on machine-learning techniques [1] operating directly on the polyhedral representation. Hybrid techniques include a part of static analysis in the classification or compression of the data gathered by the dynamic measurement for enabling the generalization of the decisions for patterns contained in programs that were not part of the training benchmark suite.

The analyzers for the profitability of a transformation sequence are critical to GRAPHITE and have to be implemented just after the translators in and out of the polyhedral form.

5.4 Integration of Array Regions

The computation of transformers and preconditions as in PIPS, will be based on the generic propagation engine [52], that has to be extended to the interprocedural mode. In intraprocedural mode, the array regions informations is not very useful, because the data dependence analysis is able to produce the same accurate results.

The extraction of more precise memory accesses in interprocedural mode can be deferred to a later stage of improvements. More improvements in the precision of the data dependence analysis can be done in parallel with the implementation of GRAPHITE as they will also benefit to other optimization passes.

5.5 Numerical Domains Common Interface

Up to now, we considered parameterized polyhedra (with integral points) as the foundation for program abstractions, representations and optimizations. In the search for faster compilation techniques or more flexible representations (beyond static control nests), we are interested in several related *numerical domains*.

A common interface for numerical domains computations has been designed in the APRON project [6] that gathers several members of the static analysis community in France: École des mines de Paris, École normale supérieure, École polytechnique, Vérimag, and IRISA. The goal of this interface is to ease the use of different numerical domains libraries with minimal code changes. Several existing libraries that implement intervals, octagons, polyhedra, linear and polynomial equalities, polynomial inequalities, etc., have been considered during the design, and a reduced number of common operations have been retained for the common interface: these operations are those that occur among all the domains, i.e. construction, meet, join, projection, etc., for which every numerical domain library is providing an implementation.

One of the main goals to the integration of this common interface in GCC is to ease the integration of new developments from the static analysis community. This interface is just a contract between the user and the implementors of the numerical domains, as the interface does not include the code of the underlying libraries: it is just a guarantee that a new numerical domain library will provide the basic operations. For this reason we will have to either consider the inclusion of the numerical domains libraries in the core of GCC, or add a new dependence on some library developed aside. In both cases there is an overhead to the inclusion. We consider the integration of the APRON common interface only as a long term project, as we can use the OMEGA library for precise operations, and implement on the side the missing specialized algorithms provided by other libraries.

In the following, we separately describe some of the libraries that contain the main functionalities needed for the GRAPHITE project.

5.5.1 OMEGA Library

The OMEGA Library [56] has been developed for solving and reducing Presburger arithmetic formulas. OMEGA is known to be expressive, but it is doubly exponential in the worst case (and often exponential in practice). This library has been already integrated to GCC, and will also be part of the APRON common interface, but it cannot (alone) face the complexity of polyhedral code generation and dependence analysis. Some specialized algorithms that are faster in practice are used: as for example the algorithm from the PIP library.

5.5.2 PIP Library

The PIP library [29] contains a specialized algorithm to compute affine objective function or lexicographical minima (or maxima) in convex polyhedra. The main algorithms of this library can be contributed to GCC as a refinement for the operations that use OMEGA.

5.5.3 Octagons

A library that provides a domain for octagons has been implemented by Antoine Miné [50, 51]. Its use in GRAPHITE would be just experimental, yet has the potential to be a local multi-criteria optimum in terms of accuracy, expressiveness (as a program representation vehicle) and compilation speed. We wish to conduct active research in this area, yet it is not on the critical path.

5.5.4 Specific Algorithms for Polyhedra

There are some libraries that implement specialized algorithms that we will consider for the reduced computation cost: we will consider the integration of the Barvinok library [65] to count in polynomial time the number of points in integer polyhedra, but also some missing parts of the PolyLib.

5.6 Maintenance of Components

The objective is to minimize the effort needed to implement and to maintain the code: smallest number of lines of code, fast algorithms specialized to compilation, rewrite some existing code, clean up, etc, as for the integration of OMEGA. It will also be interesting to benefit from the existence of active communities that develop some of the abstract numerical domains, and create dependences on outer libraries when their license is compatible.

More concretely, all existing code for the WRaP-IT project is licensed under the (L)GPL. The code is mostly implemented in C, except for parts of WRaP-IT (C++ and domain-specific transformation language). As all this code is specific to the internal representations of the compiler, it will be integrated to GCC. This will be the most costly part, in number of lines of code, to be integrated in GCC.

The analysis of the profitability will be based on libraries developed aside, and will contain fewer lines of code. The APRON library will be licensed under LGPL, and the libraries that will work within this framework are either in the public domain, as the OMEGA library, or under GPL, as the Parma PolyLib [7], the PolyLib, and Polka. We propose to keep all these libraries out of the core of GCC, for taking profit of their active communities. When one of the libraries is not maintained anymore, as in the case of OMEGA, there are several ways to handle the changes: the first solution is to host the changes on savannah as a new project for the library, the other solution is to integrate the code to an active library, or as a last resort, to integrate the code to GCC. For instance the PIP library is a short, a few thousand lines of code, specialized algorithm that could be integrated in the PolyLib.

5.7 Other Loop Optimization Modules

The heavy infrastructure of the loop-nest optimizer can be implemented independently of GRAPHITE, such as privatization, array renaming, flattening of arrays, automatic parallelization, etc. However some of these transformations could use the cost models and validity analyses developed for GRAPHITE.

6 Conclusion

GRAPHITE is the first production compiler project to implement a polyhedral loop-nest optimizer. It has a strong potential for existing architectures, and even more, as a framework to improve the productivity of compiler construction for future and emerging processors.

From a research perspective, GRAPHITE will raise several unsolved problems.

- Polyhedral frameworks only offer a limited support for interprocedural or irregular control flow. Solving this problem would lead to a dramatic extension of the potential of the polyhedral model.
- Is there a good tradeoff in expressiveness and algorithmic complexity? Are the octagons precise enough to represent enough common cases?
- What is the future of cost models and heuristics, based on polyhedral description of target machines, static and feedback-directed performance estimation?

References

[1] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *4th Annual International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006.

[2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *ACM Supercomputing'00*, May 2000.

[3] R. J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, october 1987.

[4] R. J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'91)*, pages 39–50, June 1991.

[6] Apron: Numerical program analysis. `http://www.cri.ensmp.fr/apron/`.

[7] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In M. V. Hermenegildo and

G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium, LNCS 2477*, pages 213–229, 2002.

[8] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[9] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[10] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles, France, Feb. 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[11] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. J. of Parallel Programming*, 28(3):213–243, June 2000.

[12] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. of Parallel and Distributed Computing*, 40:210–226, 1997.

[13] 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.

[14] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.

[15] D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.

[16] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[17] F. Chow. Maximizing application performance through interprocedural optimization with the pathscale eko compiler suite. `http://www.pathscale.com/whitepapers.html`, Aug. 2004.

[18] P. Clauss and V. Loechner. Parametric Analysis of Polyhedral Iteration Spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP'96*. IEEE Computer Society, August 1996.

[19] A. Cohen. *Program Analysis and Transformation: from the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, France, Dec. 1999.

[20] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'04*, number 3149 in LNCS, pages 292–303, Pisa, Italy, Aug. 2004. Springer-Verlag.

[21] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, Apr. 1995.

[22] J.-F. Collard. *Reasoning About Program Transformations*. Springer-Verlag, 2002.

[23] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. of Supercomputing*, 2002.

[24] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5^{th}ACM Symp. on Principles of Programming Languages*, pages 84–96, Jan. 1978.

[25] B. Creusillet. *Array Region Analyses and Applications*. PhD thesis, École Nationale Supérieure des Mines de Paris (ENSMP), Paris, France, Dec. 1996.

[26] B. Creusillet and F. Irigoin. Interprocedural array region analyses. *Int. J. of Parallel Programming*, 24(6):513–546, Dec. 1996.

[27] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

[28] C. Eisenbeis and J.-C. Sogno. A general algorithm for data dependence analysis. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 292–302, Washington, D. C., United States, 1992. ACM Press.

[29] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sept. 1988.

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

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

[32] G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *11thWorkshop on Languages and Compilers for Parallel Computing*, LNCS, Washington DC, July 2002. Springer-Verlag.

[33] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[34] 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. *Int. J. of Parallel Programming*, 2006. 57 pages. Accepted for publication.

[35] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, New York, june 1991.

[36] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[37] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EuroPar'95*, volume 966 of *LNCS*, pages 315–326. Springer-Verlag, 1995.

[38] F. Irigoin, P. Jouvelot, and R. Triolet. Overview of the PIPS project. In P. Feautrier and F. Irigoin, editors, *2ndIntl. Workshop on Compilers for Parallel Computers*, pages 199–212, Paris, Dec. 1990.

[39] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Int. Conf. on Supercomputing (ICS'91)*, Cologne, Germany, June 1991.

[40] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems.

Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[41] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

[42] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[43] X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 International Conference on Parallel Processing*, pages 204–211, St. Charles, august 1990.

[44] L. Lamport. The parallel execution of do loops. *Communications of ACM*, 17(2):83–93, 1974.

[45] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, Apr. 1994.

[46] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24thACM Symp. on Principles of Programming Languages*, pages 201–214, Paris, France, jan 1997.

[47] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 102–112, 2001.

[48] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *J. of Supercomputing*, 21(1):37–76, Jan. 2002.

[49] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14, New York, NY, USA, 1991.

[50] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[51] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. (to appear) `http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf`.

[52] D. Novillo. A propagation engine for gcc. In *Proceedings of the 2005 GCC Developers Summit*, pages 175–184, 2005. `http://www.gccsummit.org/2005`.

[53] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, Pittsburgh, Pennsylvania, Nov. 2004. 15 pages.

[54] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model*. Number 1132 in LNCS. Springer-Verlag, 1996.

[55] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, Aug. 1991.

[56] W. Pugh. The omega test: a fast and practical integer programming algorithm for

dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[57] W. Pugh. Uniform techniques for loop optimization. In *ACM Int. Conf. on Supercomputing (ICS'91)*, pages 341–352, Cologne, Germany, June 1991.

[58] W. Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[59] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, Oct. 2000.

[60] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[61] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, UK, 1986.

[62] R. Triolet, P. Feautrier, and P. Jouvelot. Automatic parallelization of fortran programs in the presence of procedure calls. In *Proc. of the 1^{st}European Symp. on Programming (ESOP'86)*, number 213 in LNCS, pages 210–222. Springer-Verlag, Mar. 1986.

[63] P. Tu and D. Padua. Automatic array privatization. In *6^{th}Workshop on Languages and Compilers for Parallel Computing*, number 768 in LNCS, pages 500–521, Portland, Oregon, Aug. 1993.

[64] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, Mar. 2006. Springer-Verlag.

[65] S. Verdoolaege, K. Woods, M. Bruynooghe, and R. Cools. Computation and manipulation of enumerators of integer projections of parametric polytopes. Technical Report CW 392, Katholieke Universiteit Leuven, Dept. of Computer Science, 2005. `http://www.kotnet.org/~skimo/barvinok/`.

[66] H. L. Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[67] F. Vivien. *Détection de parallélisme dans les boucles imbriquées*. PhD thesis, ENS - Lyon, 1997.

[68] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, Aug. 1992. Published as CSL-TR-92-538.

[69] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[70] M. Wolfe and C. W. Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, 1992.

[71] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[72] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.