Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen and Nicolas Vasilache
ALCHEMY Group, INRIA FUTURS and Paris-Sud University
firstname.lastname@inria.fr

Abstract

Emerging microprocessors offer unprecedented parallel computing capabilities and deeper memory hierarchies, increasing the importance of loop transformations in optimizing compilers. Because compiler heuristics rely on simplistic performance models, and because they are bound to a limited set of transformations sequences, they only uncover a fraction of the peak performance on typical benchmarks. Iterative optimization is a maturing framework to address these limitations, but so far, it was not successfully applied complex loop transformation sequences because of the combinatorics of the optimization search space.

We focus on the class of loop transformation which can be expressed as one-dimensional affine schedules. We define a systematic exploration method to enumerate the space of all legal, distinct transformations in this class. This method is based on an upstream characterization, as opposed to state-of-the-art downstream filtering approaches. Our results demonstrate orders of magnitude improvements in the size of the search space and in the convergence speed of a dedicated iterative optimization heuristic.

1. Introduction

Feedback-directed and iterative optimizations have become essential defenses in the fight of optimizing compilers to stay competitive with hand-optimized code: they freshen the static information flow with dynamic properties, adapting to complex architecture behaviors, and compensating for the inaccurate single-shot of model-based heuristics. Whether a single application (for client-side iterative optimization) or a reference benchmark suite (for in-house compiler tuning) are considered, the two main trends are:

- tuning or specializing an individual heuristic, adapting the profitability or decision model of a given transformation [35];
- tuning or specializing the selection and parameterization of existing (black-box) compiler phases [36, 1].

This paper takes a more offensive position in this fight. To avoid diminishing returns in tuning individual phases or combinations of those, we collapse multiple optimization phases into a single, unconventional, iterative search algorithm. By construction, the search space we explore encompasses *all legal program transformations* in a particular class. Technically, we consider the whole class of loop nest transformations that can be modeled as *one-dimensional schedules* [14], a significant leap in model and search space complexity compared to state-of-the-art applications of iterative optimization. We make the following contributions:

- we statically construct the (infinite in general) optimization space of *all*, arbitrarily complex, arbitrarily long sequences of loop transformations that can be expressed as one-dimensional affine schedules (using a polyhedral abstraction);
- this search space is built free of illegal and redundant transformation sequences, avoiding them altogether at the very source of the exploration;
- we demonstrate multiple orders of magnitude reduction in the size of the search space, compared to filtering approaches on loop transformation sequences, or state-of-the-art affine scheduling;
- such smaller spaces are amenable to fast-converging, operation research algorithms, allowing to compute the exact size of the space, and sometimes to traverse it exhaustively;
- our approach is compatible with acceleration techniques for feedback-directed optimization, including machine-learning techniques to focus the search to a narrow set of promising transformations;
- our source-to-source transformation tool yields significant performance gains on top of a heavily tuned, aggressive optimizing compiler;
- we provide evidence of the intricacy of the optimal code, a confirmation that building a predictive model for loop transformation *sequences* seems out of reach.

2. Related Work

Iterative compilation aims at selecting the best parameterization of the optimization chain, for a given program

or for a given application domain. It typically affects optimization flags (switches), parameters (e.g., loop unrolling, tiling), phase ordering, the heuristic itself, or the hybridation of multiple heuristics [10, 8, 3, 25, 1, 30, 35, 9, 24]

This paper studies a different search space: instead of relying on existing compiler options to transform the program, we statically construct a set of candidate program versions, considering the distinct result of all legal transformations in a particular class. Building an actual optimization phase out of this search space is much easier than from the composition of multiple search spaces arising from short-sighted, local transformations. Our method is also complementary to other forms of iterative optimization which address the orchestration of existing heuristics. Furthermore, it is completely independent from the compiler back-end.

Because iterative compilation relies on multiple, costly "runs" (including compilation and execution), the current emphasis is on improving the profiling cost of each program version [25, 17], or the total number of runs, using, e.g., genetic algorithms [23] or machine learning [1, 9]. Our heuristic is tuned to the rich mathematical properties of the underlying *polyhedral* model of the search space, and exploits the regularity of this model to reduce the number of runs. Combining it with machine learning techniques seems promising and is the subject of our ongoing work.

The polyhedral model is a well studied, powerful mathematical framework to represent loop nests and their transformations, overcoming the limitations of classical, syntax-driven models. Many studies have tried to assess a predictive model characterizing the best transformation within this model, mostly to express parallelism [26, 15] or to improve locality [41, 12, 29]. We show that such models do not scratch the complexity of the target architecture and the interference of the back-end compiler phases, yielding sub-optimal results even on simple kernels.

Iterative compilation associated to the polyhedral model is not a very common combination. To the best of our knowledge, only Long et al. tried to define a search space based on this model [28, 27], using the Unified Transformation Framework [20] and targeting Java applications. Long's search space includes a potentially large number of redundant and/or illegal transformations, that need to be discarded after a legality check, and the fraction of distinct and legal transformations decreases exponentially to zero with the size of program to optimize. On the contrary, we show how to build and to take advantage of a search space which, by construction, contains no redundant and no illegal transformation.

3. Generating a Variety of Program Versions

Program restructuring is usually broken into sequences of primitive transformations. In the case of loops, typical primitives are the loop *fusion*, loop *tiling*, or loop *interchange* [2]. This approach has severe drawbacks. First, it is difficult to decide the completeness of a set of directives and to understand their interactions. Many different sequences lead to the same target code and it is typically impossible to build an exhaustive set of candidate transformed programs in this way. Next, each basic transformation comes with its own application criteria such as legality check or pattern-matching rules. For instance it is unlikely that loop fusion would be applied by a compiler if the bounds of the original loops do not match (while this may be the result of a former transformation in the sequence). Finally, long sequences of transformations contribute to code size explosion, polluting instruction cache and potentially forbidding further compiler optimizations.

Instead of reasoning on transformation sequences, we look for a representation where composition laws have a simple structure, with at least the same expressiveness as classical transformations, but without conversions to or from transformation descriptions based on sequences of primitives. To achieve this goal, we used an algebraic representation of both programs and transformations. This is the so-called *polyhedral representation*; it is introduced in Section 3.1. We will focus on a sub-class of transformations that can be modeled through *one-dimensional schedules*; this class is described in Section 3.2.

3.1. An Algebraic Program Representation

Only parts of the program, called *Static Control Parts* (SCoP), can be represented algebraically in the polyhedral model. Roughly, a SCoP is a maximal set of consecutive instructions such that: the only allowed surrounding control structures are `for` loops and `if` conditionals, loop bounds and conditionals are affine functions of the surrounding loop iterators and the global parameters.

The significance of SCoPs has been widely discussed by Girbal et al. [18], showing that they capture a large portion of the computation time of scientific and signal processing applications.

In such a program class, semantic information can be represented as polyhedra of integer points. For instance, let us consider the `matvect` kernel in Figure 1.

```
    for (i = 0; i <= n; i++) {
R   |  s[i] = 0;
    |  for (j = 0; j <= n; j++)
S   |  |  s[i] = s[i] + a[i][j] * x[j];
    }
```

Figure 1. `matvect` **kernel**

Instruction R is enclosed by a single loop iterating on i. Its *iteration vector* $\vec{x}_R$ is the vector (i). Iterator i takes values between 0 and n, hence the polyhedron containing all

the values successively taken by i is $\mathcal{D}_R : \{i \mid 0 \le i \le n\}$. Intuitively, to each point of the polyhedron corresponds an execution of instruction R, called an *instance*, where the value of the loop iterator i is the corresponding point coordinates in the polyhedron. With a similar reasoning we can express the iteration domain of instruction S: $\vec{x}_S = \binom{i}{j}$. The polyhedron representing its iteration domain is $\mathcal{D}_S : \{i, j \mid 0 \le i \le n \wedge 0 \le j \le n\}$.

In the remainder, we use a matrix representation with so-called *homogeneous* coordinates to express systems of affine (in)equalities (the extra column expresses the affine part of every (in)equality). For instance, for the iteration domain of R, we get:

$$\mathcal{D}_R : \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} \ge \vec{0}$$

Each statement in a SCoP will be represented using its iteration domain and a set of data references. For our purpose, we consider only array accesses with affine subscript functions of outer loop iterators and global parameters (scalars may be seen as degenerate cases of arrays). In this way, array references can be expressed using matrices, for instance the reference to array a in Figure 1 is $a[i][j]$ or $a[f(\vec{x}_S)]$ with

$$f(\vec{x}_S) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

Other kinds of array references have to be modeled conservatively. Pointers arithmetic is forbidden (except when translated by a former restructuring pass to array-based references [16]) and function calls have to be inlined.

3.2. One-Dimensional Schedules

A *schedule* is a function which associates a logical execution date (a timestamp) to each execution of a given statement. In the target program, statement instances will be executed according to the increasing order of these execution dates. Two instances (possibly associated with distinct statements) with the same timestamp can be run in parallel. This date can be either a scalar (we will talk about one-dimensional schedules), or a vector (multidimensional schedules). We only consider *affine* schedules for decidability reasons.

A *one-dimensional* schedule, if it exists, expresses the program as a single *sequential* loop, possibly enclosing one or more *parallel* loops. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops. Affine schedules have been extensively used to design systolic arrays [33] and in automatic parallelization programs [14, 11, 19], then have seen many other applications.

In this study, we focus on affine one-dimensional schedules: given a statement S, it is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written

$$\theta_S(\vec{x}_S) = T \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$

where T is a constant row matrix. Such a representation is much more expressive than sequences of primitive transformations, since a single one-dimensional schedule may represent a potentially intricate and long sequence of any of the transformations shown in Figure 2. All these transformations can be represented as a partial order in the space of all instances for all statements, and such orderings may be expressed with one-dimensional scheduling functions [40].

There exist robust and scalable algorithms and tools to reconstruct a loop nest from a polyhedral representation (i.e., from a set of affine schedules) [21, 32, 5]. We will thus generate transformed versions of each SCoP by exploring its legal, distinct affine schedules, regenerating a loop nest program every time to profile its effective performance.

4. Building The Search Space

In general, restructuring a program will change its semantics. When a transformation preserves the original program semantics, we will say that it is *legal*. Previous works on iterative optimization using a polyhedral representation ensure this property by checking, after computing a transformation, whether it is legal or not [28, 27] (non-iterative optimization algorithms use either a similar approach [22], either consider programs simple enough that nearly every transformation is possible [42]). This results in considering huge search spaces, since every illegal or redundant solutions have to be checked, and to a significant computation overhead corresponding to each legality check (typically most of them stating that the transformation must not be applied). Such an approach cannot scale since the number of redundant and/or illegal transformations grows exponentially faster than the number of different and legal transformations with the size of the input program.

To overcome those issues, we build a search space which, by construction, encompasses all legal program transformations in the class of one-dimensional schedule. The following sections present this search space, recalling how we represent data dependences in our algebraic representation in Section 4.1, then constructing the space itself in Section 4.2, thanks to a deep result in linear algebra.

4.1. Data Dependence Representation

Two statements instances are in *dependence relation* if they access the same memory cell and at least one of these

Transformation	Description
`reversal`	Changes the direction in which a loop traverses its iteration range
`skewing`	Makes the bounds of a given loop depend on an outer loop counter
`interchange`	Exchanges two loops in a perfectly nested loop, a.k.a. `permutation`
`peeling`	Extracts one iteration of a given loop
`index-set splitting`	Partitions the iteration space between different loops
`shifting`	Allows to reorder loops
`fusion`	Fuses two loops, a.k.a. `jamming`
`distribution`	Splits a single loop nest into many, a.k.a. `fission` or `splitting`

Figure 2. Possible Transformations Embedded in a One-Dimensional Schedule

accesses is a write operation. For a program transformation to be correct, it is necessary to preserve the original execution order of such statement instances and thus to know precisely the instance pairs in dependence relation. In the algebraic program representation depicted in section 3.1, it is possible to characterize exactly the set of instances in dependence relation in a very synthetic way.

Three conditions have to be satisfied to state that a statement instance $R(\vec{x}_R)$ depends on a statement instance $S(\vec{x}_S)$. (1) They must refer the same memory cell, which can be expressed by equating the subscript functions of a pair of references to the same array. (2) They must be actually executed, i.e. $\vec{x}_S$ and $\vec{x}_R$ have to belong to their corresponding iteration domains. (3) $S(\vec{x}_S)$ is executed before $R(\vec{x}_R)$ in the original program. Each of these three conditions may be expressed using affine inequalities (see section 3.1, or [2] for more details). It leads that exact sets of instances in dependence relation can be represented using affine inequality systems.

For instance, if we consider the `matvect` kernel in Figure 1, dependence analysis gives two dependence relations: instances of statement S depending on instances of statement R (e.g., R produces values used by S), $\delta_{R,S}$, and similarly, $\delta_{S,S}$.

Dependence relation $\delta_{R,S}$ does not mean that all instances of R and S are in dependence (for all values of $\vec{x}_R$ and $\vec{x}_S$); in fact, there is only a dependence if $i_R = i_S$. We can then define a *dependence polyhedron*, being a subset of the Cartesian product of the iteration domains, containing all the values of i_R, i_S and j_S for which the dependence exists. We can write this polyhedron in matrix representation (the first line represents the equality $i_R = i_S$, the two next ones the constraint that (i_R) have to belong to the iteration domain of R and similarly, the four last lines states that (i_S, j_S) belongs to the iteration domain of S):

$$\mathcal{D}_{R,S} : \left[\begin{array}{ccccc} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{array} \right] \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \begin{array}{c} = 0 \\ \hline \geq \vec{0} \end{array}$$

4.2. Legal Transformation Space

The data dependence analysis gives the exact information on which statement instance pairs have to respect their relative original execution order. Let R and S be two statements. Each (integral) point of the dependence polyhedron $\mathcal{D}_{R,S}$ represents a value of the iteration vectors $\vec{x}_R$ and $\vec{x}_S$ where the dependence needs to be satisfied. It is possible to express the set of affine, non-negative functions over $\mathcal{D}_{R,S}$ thanks to the affine form of the Farkas lemma [14]:

Lemma 1 *(Affine form of Farkas Lemma [34]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\,\vec{x} + \vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}^T(A\vec{x} + \vec{b}), \text{ with } \lambda_0 \geq 0 \text{ and } \vec{\lambda}^T \geq \vec{0}.$$

λ_0 and $\vec{\lambda}^T$ are called Farkas multipliers.

In order to satisfy the dependence, the schedules have to satisfy the precedence condition $\theta_R(\vec{x}_R) < \theta_S(\vec{x}_S)$, for each point of $\mathcal{D}_{R,S}$. So one can state that

$$\Delta_{R,S} = \theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) - 1$$

must be non-negative everywhere in $\mathcal{D}_{R,S}$. Since we can express the set of affine non-negative functions over $\mathcal{D}_{R,S}$, the set of legal schedules satisfying the dependence $\delta_{R,S}$ is given by the relation

$$\Delta_{R,S} = \lambda_0 + \vec{\lambda}^T \left(D_{R,S} \begin{pmatrix} \vec{x}_R \\ \vec{x}_S \end{pmatrix} + \vec{d}_{R,S} \right) \geq 0$$

where $D_{R,S}$ is the constraint matrix representing the polyhedron $\mathcal{D}_{R,S}$ over $\vec{x}_R$ and $\vec{x}_S$, and $\vec{d}_{R,S}$ is the scalar part of these constraints.

Let us go back to the `matvect` example in Figure 1. The two prototype affine schedules for R and S are:

$$\begin{aligned}\theta_R(\vec{x}_R) &= t_{1_R}.i_R + t_{2_R}.n + t_{3_R}.1\\ \theta_S(\vec{x}_S) &= t_{1_S}.i_S + t_{2_S}.j_S + t_{3_S}.n + t_{4_S}.1\end{aligned}$$

Using the previously defined dependence representation, we can split the system into as many inequalities as there are independent variables, and equate the coefficients in both sides of the equation. For dependence $\delta_{R,S}$ we have

$$\left\{\begin{array}{rcrcl} i_R & : & -t_{1_R} & = & \lambda_1 + \lambda_2 - \lambda_3\\ i_S & : & t_{1_S} & = & -\lambda_1 + \lambda_4 - \lambda_5\\ j_S & : & t_{2_S} & = & \lambda_6 - \lambda_7\\ n & : & t_{3_S} - t_{2_R} & = & \lambda_3 + \lambda_5 + \lambda_7\\ 1 & : & t_{4_S} - t_{3_R} - 1 & = & \lambda_0 \end{array}\right.$$

where λ_x is the Farkas multiplier attached to the xth line of $D_{R,S}$.

This system expresses all the constraints a schedule have to respect according to the dependence $\delta_{R,S}$. In order to get a tractable set of constraints on the schedule coefficients, we need to solve the system, with for example the Fourier-Motzkin projection algorithm [14]. If there is no solution, then no affine one-dimensional schedule is possible for this dependence.

If we build then solve the system for the dependence $\delta_{R,S}$, we obtain a polyhedron $\mathcal{T}_{R,S}$, by projecting the λ dimensions on the t ones (the corresponding schedule variables of R and S). This polyhedron represents the set of legal values for the schedule coefficients, in order to satisfy the dependence. To build the set of legal schedule coefficients for the whole program, we have to build the intersection of each polyhedron obtained for each dependence. The result is a global polyhedron $\mathcal{T}$, with as many dimensions as there are schedule coefficients for the SCoP, the intersection of the constraints obtained for each dependence. With this method, the transitivity of the dependence relation is preserved in the global solution but all systems are built and solved one dependence at a time. In fact, the computation of the legal space can be done simultaneously with the dependence analysis. The intersection operation implicitly extends the dimensionality of polyhedra to the dimensionality of $\mathcal{T}$, and sets the missing dimensions as unconstrained. So we have for k dependences:

$$\mathcal{T} = \bigcap_k \mathcal{T}_k$$

Intuitively, to each (integral) point of $\mathcal{T}$ corresponds a different schedule for the original program, i.e., a different program version (or also a valid, distinct transformation sequence). Enumerating points in this polyhedron can be done by polyhedral code generation algorithms, but even though our problem lies into the (simpler) convex case, they may not scale over thirty to forty dimensions [4, 37] because of the intrinsic combinatorics of characterizing the polyhedron's integral hull. Fortunately, our problem happens to be much simpler than the "static" loop nest generation one: we only need to "dynamically" enumerate every integral point which respects the set of constraints provided by $\mathcal{T}$. We may thus incrementally pick a dimension then *pick an integer* in the polyhedron's projection onto this dimension. This incremental method combines low-complexity projections with the Fourier-Motzkin algorithm and simple enumerations of dense polyhedra.

5. Practical Search Space

The legal one-dimensional schedule space for a given SCoP as described in section 4.2 is possibly infinite. For instance it is easy to see that if there is no data dependence at all, every value of the schedule coefficients is possible. It is necessary to bound this space in such a way that an exhaustive scan becomes possible. Bounding the space will remove some possible program transformations. We have to ensure we remove only the less interesting solutions for performance.

We can distinguish two families of coefficients in the schedule expressions, (1) iterator coefficients, (2) parameter and constant coefficients. Each family will provide a specific contribution to the global program transformation [6]. The iterator coefficients will impact on loop structure and bounds (*skewing*-like transformations for instance) while parameters and constant will impact on loop ordering and statement ordering within a loop (*shifting*-like transformations for instance). It follows, while the order of magnitude of coefficients values for parameters and constant do not have any influence on performance, using big iterator coefficients will result in a very high control overhead (like generation of complex loop bounds and costly modulo operation) that will waste the optimization they are potentially enabling [5]. Hence we should bound the values of the iterator coefficients with small values (we checked empirically that the bounding interval $[-1, 1]$ is wide enough most of the time).

The coefficients of the parameters and the constant have also to be bounded to avoid an infinite search space. The difference between the two bounds should be greater than the number of statements to ensure that at least every ordering of the statements within or outside loops is possible. Greater intervals will offer more possibilities, for instance to achieve more peeling transformations but a large flexibility is rarely useful in practice.

Benchmark	#Dependences	$\vec{\imath}$-Bounds	$\vec{p}$-Bounds	c-Bounds	#Schedules	#Legal	Time
h264	15	$-1,1$	$-1,1$	$0,4$	7.5×10^5	360	0.011
fir	12	$-1,1$	$-1,1$	$-1,1$	4.7×10^6	432	0.004
fft	36	$-2,2$	$-2,2$	$0,6$	5.8×10^{25}	804	0.079
lu	14	$0,1$	$0,1$	$0,1$	3.2×10^4	1280	0.005
gauss	18	$-1,1$	$-1,1$	$-1,1$	5.9×10^4	506	0.021
crout	26	$-3,3$	$-3,3$	$-3,3$	2.3×10^{14}	798	0.027
matmult	7	$-1,1$	$-1,1$	$-1,1$	1.9×10^4	912	0.003
MVT	10	$-1,1$	$-1,1$	$-1,1$	4.7×10^6	16641	0.001
locality	2	$-1,1$	$-1,1$	$-1,1$	5.9×10^4	6561	0.001

Figure 3. Search space computation

We made several tests to compare our approach, taking into account only the legal schedules, to considering every schedules and filter legal ones thanks to a legality check, as Long et al. suggests [28]. We used different compute-intensive kernel benchmarks coming from various origins and listed in Figure 3. `h264` is a fractional sample interpolation of the H.264 standard [39]. `fir` and `fft` are DSP kernels extracted from UTDSP benchmark suite [39]. `lu`, `gauss`, `crout` and `matmult` are well known mathematical kernels corresponding to LU factorization, Gaussian elimination, Crout matrix decomposition and matrix-matrix multiply. `MVT` is a kernel including two matrix-vector multiplications, one matrix being the transposition of the other. `locality` is an hand-written memory access intensive kernel. Notice our motivation is not to evaluate the performance of our schedules with respect to aggressive optimizations performed manually (like the BLAS), or by application-specific active libraries (like ATLAS or SPIRAL): we are evaluating an automatic source-to-source framework, exploring *all but only* one-dimensional schedules, and not considering any domain-specific knowledge.

These kernels are typically small, from 2 to 17 statements. They suit well the present study and allow fair comparison with present production compiler: first, they should not challenge present production compiler optimization schemes, and second, they will make it possible to achieve an exhaustive visit of our search space which is necessary to evaluate the potential of the method and to design heuristic techniques. Dealing with larger benchmarks presents some technical difficulties: first of all, every SCoP does not have a one-dimensional schedule, and the likeliness decreases with the complexity of the dependence graph; second, although we achieved a breakthrough in allowing much larger optimization spaces to be characterized and traversed, going beyond 20 to 30 array accesses breaks the scalability of our constraint simplification method (based on Fourier-Motzkin elimination), due to the hundreds of transformation coefficients to consider simultaneously. Further scalability may be achieved through algorithmic improvements in the exploitation of regularity properties in the constraints systems, and through heuristics to prioritize the most important dependences and / or to partition the problem into smaller, modular scheduling spaces. We are currently investigating these ideas.

Figure 3 summarizes the study of the search space. The first column presents the various kernel benchmarks; the second one labeled #Dependences precises the number of dependence relations for the corresponding kernel; $\vec{\imath}$-Bounds shows the iterator coefficient bounds used for search space bounding; $\vec{p}$-Bounds shows the parameter coefficient bounds; c-Bounds shows the constant coefficient bounds; #Schedules shows the total number of schedules, including illegal ones; #Legal shows the number of actual schedules in our space, i.e. the number of legal schedules; finally, Time shows the search space computation time on a Pentium 4 Xeon, 3.2GHz.

Results shows the very high benefit to work directly on a space including only legal transformations since it lowers the number of considered transformations by one to many orders of magnitude for a quite acceptable computation time. On the contrary, these results shows that without such a politic, achieving an exhaustive search is not possible even for small kernels. While these results shows profitability, it is not a demonstration of scalability, in the following we will propose to actually visit the search space exhaustively or using an heuristic way.

6. Scanning the Optimization Search Space

In previous sections, we formally recalled how to build a singular search space where each point corresponds to a distinct legal program version. We also adapted this space in such a way that a scan becomes possible in any case. In the following, we will actually visit the search space to evaluate its potential for program optimization. In section 6.1, we present our experimental setup, section 6.2 shows results on exhaustive search while section 6.7 presents a heuristic to avoid performing a large number of runs while preserving the core optimization benefits.

```
  S1(i): a[i] = i
  S2(i,j): b[j] = (b[j] - a[i]) / 2

 Original code:
  for (i = 0; i <= M; i++) {
  | S1(i);
  | for (j = 0; j <= N; j++) {
  | | S2(i,j);
  | }
  }

 Chunked code:
  for (t = 0; t <= M; t++) {
  | S1(t);
  }
  for (t = M; t <= M+N; t++) {
  | for (i = 0; i <= M; i++) {
  | | S2(i,t-M);
  | }
  }
```

```
 best transformation:
  S1(0);
  for (t = -M+1; t <= 0; t++) {
  | for (i = max(0, t+M-N-1); i <= t+M-1; i++) {
  | | S2(i,t-i+M-1);
  | }
  | S1(t+M);
  }
  for (t = 1; t <= N+1; t++) {
  | for (i = max(t+M-N-1, 0); i <= M; i++) {
  | | S2(i,t-i+M-1);
  | }
  }
```

Figure 4. Intricacy of transformed code

6.1. Experimental Setup

We implemented dependence analysis, legal transformation space construction and scanning. We used for that purpose external publicly available tools such as `PipLib`, a linear algebra tool [13] and `CLooG`, a code generator in the polyhedral model [5]. We designed our tools to be able to use them as a plugin in the future GRAPHITE GCC's polyhedral framework [31]. The experimental protocol is as follows. For each point of the search space, (1) generate the kernel code with CLooG[1] (2) add input initialization and measure tools, to produce a C compilable unit (3) Compile it provided a compiler and its optimization options (4) run the program on the target architecture and gather the results. In order to be consistent, the original code is included in this procedure starting at the second step.

We ran our experiments on an Intel workstation based on Xeon 3.2GHz, 16KB L1, 1024KB L2 caches. We used four different compilers: GCC 3.4.2, GCC 4.1.1, Intel ICC 9.0.1 and PathScale EKOPath 2.5. We used hardware counters to measure the number of cycles used by various programs. In order to avoid interferences with other programs and the system, we set the system scheduler policy to FIFO for every test. The kernel benchmark set is the one presented in section 5.

6.2. Exhaustive Space Scanning

Because our search space is only based on legal schedules, the number of solutions for kernel benchmarks is small enough to make it possible to achieve an exhaustive search in a reasonable amount of time. Figure 5 summarizes our results. The Benchmark column states the input program; the Compiler column shows the compiler used to build each program version of the search space (GCC version was 4.1.1); the Options column precises the full compiler options; the Parameters column shows the values of the global parameters (e.g., for array sizes, parameters are chosen to exceed L2 cache size); the #Improved column shows the number of version that achieves a better performance than the original program (the total number of versions is shown in Figure 3); the ID best gives the unique "identifier" of the best solution; lastly, the Speedup column gives the speedup achieved by the best solution with respect to the original program performance. On average, one second is needed to explore a point (code generation, compilation and run of the target version).

The two main results shown by this figure are, first of all, that the best program version highly depends on both compiler and compiler options. Even considering the several very best solutions, there are typically no intersection between the set of best transformations for two pairs compiler/compilation-options. Second, significant speedups are achieved thanks to the traversal of the search space, demonstrating the interest of the method for optimizing compilation. In few cases, a 0% speedup is achieved, meaning that the original code was already optimal for our experimental setup and model. In average, the method leads to a 35.4% speedup, or to 14.9% excluding the extreme results of matrix-multiply kernel which is known to be a good candidate for such study. A global constatation is the correlation between observed speedups and locality improvements and / or transformations enabled in the backend compiler by our program versions.

6.3. Intricacy of the Best Program Version

Another interesting result is the form of the best transformed programs since they typically appear to be quite complex. Most of the time, it was not possible to easily understand which part of the transformation sequence was responsible for the speedup since a significant part of the answer was related to the compiler design. We also noticed

[1] CLooG version 0.14.0, with default options

Benchmark	Compiler	Options	Parameters	#Improved	ID best	Speedup
h264	PathCC	-Ofast	none	11	352	36.1%
	GCC	-O2		19	234	13.3%
	GCC	-O3		26	250	25.0%
	ICC	-O2		27	290	12.9%
	ICC	-fast		0	N/A	0%
fir	PathCC	-Ofast	N=50000	240	72	6.0%
	GCC	-O2		259	192	15.2%
	GCC	-O3		119	289	13.2%
	ICC	-O2		420	242	18.4%
	ICC	-fast		315	392	3.4%
fft	PathCC	-O2	N=256 M=256 O=8	21	267	7.2%
	GCC	-O2		10	285	0.9%
	GCC	-O3		11	289	1.8%
	ICC	-O2		17	260	6.9%
	ICC	-fast		20	112	6.4%
lu	PathCC	-Ofast	N=1000	100	224	6.5%
	GCC	-O2		321	339	1.6%
	GCC	-O3		330	337	3.9%
	ICC	-O2		281	770	9.0%
	ICC	-fast		262	869	8.7%
gauss	PathCC	-Ofast	N=150	212	4	3.1%
	GCC	-O2		204	2	1.7%
	GCC	-O3		52	2	0.01%
	ICC	-O2		63	288	0.05%
	ICC	-fast		15	39	0.03%
crout	PathCC	-Ofast	N=150	0	N/A	0%
	GCC	-O2		132	638	3.6%
	GCC	-O3		56	628	1.7%
	ICC	-O2		37	625	0.5%
	ICC	-fast		63	628	2.9%
matmult	PathCC	-Ofast	N=250	402	283	308.1%
	GCC	-O2		318	573	243.6%
	GCC	-O3		345	143	248.7%
	ICC	-O2		390	311	56.6%
	ICC	-fast		318	641	645.4%
MVT	PathCC	-Ofast	N=2000	5652	4934	27.4%
	GCC	-O2		3526	13301	18.0%
	GCC	-O3		3601	13320	21.2%
	ICC	-O2		5826	14093	24.0%
	ICC	-fast		5966	4879	29.1%
locality	PathCC	-Ofast	N=10000, M=2000	6069	5430	47.7%
	GCC	-O2		30	5494	19.0%
	GCC	-O3		589	4332	6.0%
	ICC	-O2		3269	2956	38.4%
	ICC	-fast		4614	3039	54.3%

Figure 5. Search space statistics for exhaustive scan

that optimization algorithms based on formal representations were sometimes far away from the optimal solution. A very simple but striking example is shown in Figure 4.

The simple, supposed optimal locality transformation in our class suggests a schedule of (i) for $S1$ and $(j+n)$ for $S2$ [7], which results in maximizing the reuse of the array a. The very best schedules were in fact $(i-j)$ and $(i+j-n+1)$ (the code generated by our framework is given in Figure 4). While the supposed optimal schedules generate a speedup of 147% with $n=100$ and $m=500k$ using GCC

3.4, the very best schedules generate a speedup of 398% (with a similar number of L1 and L2 cache-misses but a heavily reduced data TLB misses).

The relation with the compiler is described further in section 6.4. Section 6.5 deals with the effect of compiler options and lastly, we discuss the performance distribution in section 6.6.

6.4. The Compiler as an Element of the Target Platform

Our iterative optimization scheme is independent from the compiler and may be seen as a higher level to classical iterative compilation. In the same way as a given program transformation may better exploit a feature of a given processor, it also may enable more aggressive options of a given compiler. Because production compilers have to generate a target code in any case in a reasonable amount of time, their optimizations are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase.

To study this behavior and estimating how a higher level iterative optimization scheme may lead to better performances, we achieved a exhaustive scan of our search space for various programs and compilers with aggressive optimization options. We illustrate our results in Figure 5, studying the matrix multiplication kernel in more details in Figure 6 (this benchmark has been extensively studied, and is a typical target of aggressive optimizations of production compilers).

We tested the whole set of legal schedules within the bounds $[-1,1]$ for all coefficients (912 points), and checked the speedup for various compilers with aggressive optimizations enabled. Matrices are 250×250 arrays of double-precision floats. We compared, for a given compiler, the number of cycles the original code took (Original) to the number of cycles the best transformation took (Best) (results are in millions of cycles).

Figure 6 shows significant speedups achieved by the best transformations for each back-end compiler. Such speedups are not uncommon when dealing with the matrix-multiplication kernel. The important point is that we do not perform any tiling (it requires multi-dimensional schedules), contrary to nearly all other works (see [42, 2] for useful references).

In general, it was possible to check using PathScale EKOPath that many optimization phases have been enabled or disabled, depending on the version generated from our exploration tool. The enabling transformation aspect of our method is brought to light with for instance the `h264` benchmark: the EKOPath compiler fuse 4 times in the original version but only once with the best found one, but was able to vectorize 3 times more with our transformation. Nevertheless it is technically hard to know precisely the contribution of the one-dimensional schedule (which has a high potential, by itself, as an optimizing transformation) with respect to the enabled compiler optimizations. In the `matmult` case, Interchanging loops on k and i is the core of the transformation embedded in all best schedules found. This drastically improves locality: for instance, with ICC -fast, the number of L1 and L2 cache-misses is comparable for the original code and the best found version, but the number of data TLB misses goes from 15M to 164k, diminishing with a similar ratio the number of floating point operations executed (the results are consistent whether the matrices are allocated with `malloc` or directly on the stack). This encourages the potential of a combination with tiling.

But more transformations are embedded in the schedules, and another striking result is the high variation of the best schedules depending on the compiler. For instance the lack of the j iterator in $\theta_{S1}(\vec{x}_{S1})$ for GCC or the lack of the n parameter $\theta_{S2}(\vec{x}_{S2})$ for ICC. These results, which are consistent with the other tested programs, emphasize the need of a compiler-dedicated transformation to achieve the best possible performance. One possible explanation is the difference between optimization phases in the different back-end compilers. Compilers have reached such a level of complexity that it is no longer possible to model the effects of downstream phases on upstream ones. Yet it is mandatory to rely on the downstream phases of a back-end compiler to achieve a decent performance, especially those which cannot be embedded naturally in the polyhedral model.

6.5. On the Influence of Compiler Options

Experiments have shown a relation between the best transformations and the compiler options. For instance, in the `matmult` kernel benchmark case with the ICC compiler used with the aggressive `-fast` option, the best transformation yields a 4.5% slowdown when it is compiled with `-O2` and compared to the best one found for this compiler option. This behavior was observed on all the tested programs. Finding the best compiler options is the subject of many research works in iterative compilation (see section 2). Studying this aspect is out of the scope of the present paper but those results are a sign that combining our method with existing iterative compilation techniques is a promising way.

6.6. Performance Distribution

Exhaustive scanning of all program versions is feasible on (small) kernels, and we can observe the complete performance distribution. Figure 7 shows this distribution for the `matmult`, `locality` and `crout` examples: `matmult` and `locality` are compiled with `GCC 4.1.1 -O2`, the first `crout` with `ICC -fast`, and the second with `GCC 4.1.1 -O3`. Each graph represents the computation time of every point in the search space as a function of its number in the scanning order. Horizontal line shows the performance

Compiler	Option	Original	Best	Schedule	Speedup
GCC 3.4.2	-O3	519	163	$\theta_{S1}(\vec{x}_{S1}) = -1$ $\theta_{S2}(\vec{x}_{S2}) = k+1$	318.4%
GCC 4.1.1	-O3	515	207	$\theta_{S1}(\vec{x}_{S1}) = -i-j+n-1$ $\theta_{S2}(\vec{x}_{S2}) = k+n$	248.7%
ICC 9.0.1	-fast	465	72	$\theta_{S1}(\vec{x}_{S1}) = -i+n$ $\theta_{S2}(\vec{x}_{S2}) = k+1$	645.4%
PathCC 2.5	-Ofast	228	79	$\theta_{S1}(\vec{x}_{S1}) = j-n-1$ $\theta_{S2}(\vec{x}_{S2}) = k$	308.1%

Figure 6. Results for the `matmult` example

of the original program: every point below this line corresponds to a more efficient program version.

Although the scanning order may be a weird choice for such representation, it shows that the performance distribution is not totally random.[2]

From these observations, we conclude that:

- in most cases, contiguous regions of similar performance can be identified;
- several transformations may be close to the best performance, but the probability to find them at random can be very low (e.g., on `locality`);
- for some benchmarks (e.g., on `matmult`), strong correlations do exist but are not easily observable without reordering the index space of the transformations (the X axis on the performance distribution figures).

The impact of the compiler on the distribution is emphasized on the `crout` example, in the third and fourth graphs of Figure 7. Here we compare, for an identical original program (hence an identical optimization search space), the distribution on `ICC -fast` and `GCC 4.1.1 -O3` on the `crout` kernel benchmark. Hence, understanding performance regularities may help to find *hot* regions in the search space, thus avoiding useless runs in low-interest regions and diminishing-return searches among nearly optimal solutions. Machine learning techniques are used to solve similar problems for classical iterative optimization problems, and seem particularly promising to achieve this goal [35, 1]. We defer the application of these approaches to a further study, dedicating this paper to the study of the mathematical properties of our model, in an attempt to pruning the search space without loosing the most interesting solutions.

[2] It is not an absurd ordering though: the scanning procedure could be seen as a very deep loop nest were the outer loop iterates on values of the first iterator coefficient of the first statement and the inner loop iterates on values of the constant coefficient of the last statement.

6.7. Heuristic Search

Since it is unpractical to explore the whole search space on real-world benchmarks, we propose a heuristic to enumerate only a high-potential sub-space, using the properties of the polyhedral model to characterize the highest potential and narrowest one.

6.7.1. Decoupling Heuristic

We represent the schedule coefficients of a statement as a three component vector:

$$\theta_S(\vec{x}_S) = (\vec{\imath}\ \vec{p}\ c) \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$

Where $\vec{\imath}$ represents the iterators coefficients, $\vec{p}$ the parameters coefficients and c the constant coefficient.

In this search space representation, two neighbor points may represent a very different generated code, since a minor change in the $\vec{\imath}$ part can drastically modify the compound transformation (a program where *interchange* and *fusion* are applied can be the neighbor of a program with none of these transformations). The most significant impact on the generated code is caused by iterator coefficients, and we intuitively assume their impact on performance will be equally important. Conversely, modifying parameters or constant coefficients is less critical (especially when one-dimensional schedules are considered). Hence it is relevant to propose an exploration heuristic centered on the enumeration of the possible combinations for the $\vec{\imath}$ coefficients.

The proposed heuristic is window-search based. It decouples iterator coefficients from the others, enabling a systematic exploration of all the possible combinations for the $\vec{\imath}$ part. At first, we do not care about the values for the $\vec{p}$ and c part (they can be chosen arbitrarily in the search space, as soon as they are compatible with the $\vec{\imath}$ sequence). The resulting subset of program versions is then filtered with respect to effective performance, keeping the top points only. Then, we repeat the systematic exploration of the possible

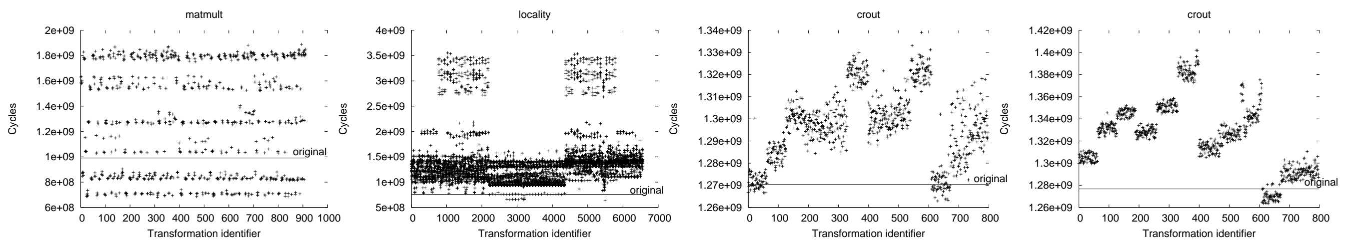

Figure 7. Performance distributions

combination of values for the $\vec{p}$ and c coefficients to refine the program transformation sequence.

The heuristic can be sketched in 5 steps.

1. Build the set of all different possible combinations of coefficients for the $\vec{\imath}$ part of the schedule, inside the set of all legal schedules. Choose $\vec{p}$ and c at random in the space, according to the $\vec{\imath}$ part.
2. For each schedule in this set, generate and instrument the corresponding program version and run it.
3. Filter the set of schedules by removing those associated with a run time more than x% slower than the best one (combined with a bound on the limit of selected schedules).
4. For each schedule in the remaining set, explore the set of possible values for the $\vec{p}$ and c part (inside the set of all legal schedules) while the $\vec{\imath}$ part is left unmodified.
5. Select the best schedule in this set.

6.7.2. Discussion

Figure 8 details a run of our decoupling heuristic (with a filtering level of 5% and a static limit of 10 points per coefficient type, see below), and compares it with a plain random search for three of our benchmarks. It shows the relative percentage of the best speedup achieved as a function of the number of iterative runs. The decoupling heuristic (the *DH* plot) yields much faster convergence, bringing to light the correlation between the speedup and the $\vec{\imath}$-coefficients. On these tested examples, one may achieve over 98% of the maximum speedup within less than 20 iterations.

On the other hand, we observed the heuristic behavior to be comparable to a full random driven approach (the *R* plot), as Figure 8 shows for the `matmult` kernel. Not surprisingly, as soon as the density of good transformations is large, a random space scan may converge faster than our enumeration-based method. For the `MVT` kernel, even if there is a large set of improved versions in the search space, the low density of good ones is emphasized by the poor convergence of the random-driven approach.

A more important problem is the scalability to larger SCoPs. To prevent the possibly large set of legal values for the $\vec{\imath}$ coefficients, it is possible to:

1. impose a static or dynamic limit to the number of runs, which should be coupled to an exploration strategy starting with coefficients as close as possible to 0 (remember 0 may not correspond to any legal schedule);
2. replace an exhaustive enumeration of the $\vec{\imath}$ combinations by a limited set of random draws in the $\vec{\imath}$ space.

The choice between the exhaustive, limited or random exploration of the $\vec{\imath}$ space can be heuristically determined with regards to the size of the original SCoP (this size usually gives a good intuition of the search space size order of magnitude).

7. Future Work

Affine multidimensional schedules It is always possible to find a multidimensional affine schedule to a SCoP, while a one-dimensional schedule may not exists. Unfortunately, the generalization of our method to multidimensional schedules leads to a well known combinatorial barrier: if there is exactly one way to choose the set of dependences to satisfy in the one-dimensional case (they must all be satisfied in one dimension, i.e. in one set), there is a combinatorial way to choose the sets as soon as there is more than one dimension. Feautrier proposed a greedy algorithm to solve the maximal set of dependences at a given depth, and increment the depth if unresolved dependences remain [15]. This would give us the minimal sequential depth of the schedule [38], but the combinatorics remains if we want to explore all the legal schedules.

Parallelism The polyhedral model has been designed to express in a natural way parallelism inside loop nests. Our study was only applied to monoprocessor machines, but it is a short term assignment to exploit this parallelism in a state-of-the-art shared-memory system. We need to slightly modify the code generation phase in order to generate an OpenMP-equipped C code.

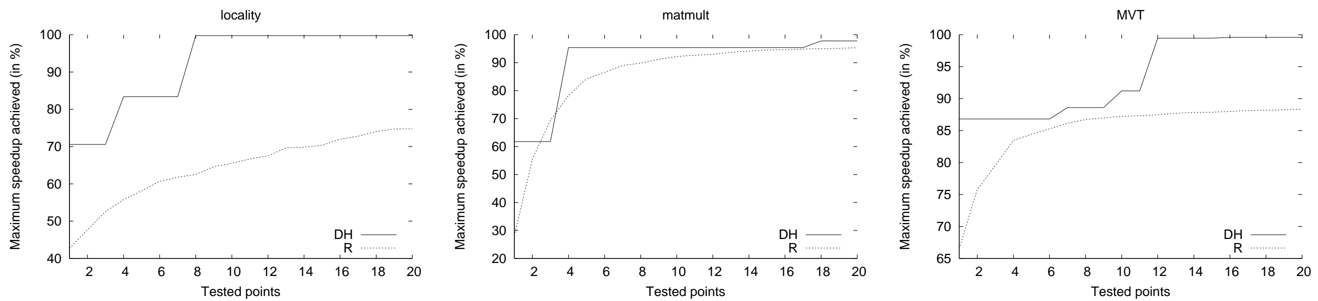

Figure 8. Comparison between the random and the decoupling heuristics

8. Conclusion

Iterative and empirical search techniques are some of the last hopes to let compilers harness the complexity of modern processors and hide it from the application programmers. We focus on loop transformations because they are both critically important for performance and very hard to drive in an optimizing compiler.

Iterative loop nest optimization is intrinsicly difficult because of the large number of distinct, legal transformations for a given program, and it is complicated by the inability of classical loop transformation frameworks to statically characterize this set. So far, all attempts have relied on a separate filtering step to remove redundant and/or illegal candidate transformation from the search space. Our experiments show that such approaches are likely to be impractical, even for tiny kernels of a few lines of code.

On the contrary, we propose an algorithm to build and traverse the whole set of distinct, legal affine one-dimensional schedules for a program, that is the expression of *every legal combination* of transformations for this class of schedules that result in *distinct program versions*. On small kernels, our early experiments demonstrate the ability to discover the wall-clock *optimal schedule*, thanks to an *exhaustive exploration* of that space, given a compiler, target architecture and data set. To our knowledge, this is the first time such a space is explored.

It is expected that a systematic exploration will not scale to large programs, or when multi-dimensional schedules are considered. We propose a heuristic-driven method to address the scalability problem, by doing a partial and focused enumeration of the initial search space. Our study also contributes key observations about the performance distributions in the transformation space, a first step towards combining our search space construction and enumeration approach with more generic machine learning or empirical search techniques.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proc. of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, New York, 2004.

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loop. In *(PPoPP'91)*, pages 39–50, June 1991.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[6] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, October 2003.

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

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

[9] J. Cavazos, J. E. Moss, and M. F. P. O'Boyle. Hybrid optimizations: Which optimization algorithm to use. In *(CC'06)*, Vienna, Austria, Apr. 2006.

[10] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *2nd Workshop on Feedback-Directed Optimization*, Israel, November 1999.

[11] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[12] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Comput.*, 54(10):1242–1257, 2005.

[13] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[14] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[15] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.

[16] B. Franke and M. O'Boyle. Array recovery and high level transformations for dsp applications. In *CPC'10 International Workshop on Compilers for Parallel Computers*, pages 29–38, Amsterdam, January 2003.

[17] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 29–46, Barcelona, Nov. 2005. Springer-Verlag.

[18] 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), 2006.

[19] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, march 2004.

[20] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical report, College Park, MD, USA, 1993.

[21] 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.

[22] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, june 1997.

[23] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, San Diego, California, USA, 2003. ACM Press.

[24] P. Kulkarni, W. Zhao, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, Y. Paek, and D. Jones. Vista: Vpo interactive system for tuning applications. *ACM Transactions on Embedded Computing Systems*. To appear.

[25] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

[26] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, New York, NY, USA, 1997. ACM Press.

[27] S. Long and G. Fursin. Systematic search within an optimisation space based on unified transformation framework. *IJCSE International Journal of Computational Science and Engineering*. To appear.

[28] S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 137–144, Washington, DC, USA, 2005. IEEE Computer Society.

[29] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[30] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proc. of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.

[31] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, June 2006.

[32] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.

[33] P. Quinton and V. V. Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1(2):95–113, october 1989.

[34] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[35] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[36] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[37] 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.

[38] F. Vivien. On the optimality of Feautrier's scheduling algorithm. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 299–308, London, UK, 2002. Springer-Verlag.

[39] T. Wiegand, G. Sullivan, and A. Luthra. Itu-t rec. h.264 – iso/iec 14496-10 avc - final draft. Technical report, Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, May 2003.

[40] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of computer science, Stanford University, California, 1992.

[41] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.

[42] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.