Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time

Louis-Noël Pouchet[1,3], Cédric Bastoul[1,3], Albert Cohen[1,3] and John Cavazos[2,3]

[1] ALCHEMY Group, INRIA Futurs and Paris-Sud University
{firstname.lastname}@inria.fr
[2] Dept. of Computer & Information Science, University of Delaware
cavazos@cis.udel.edu
[3] High Performance for Embedded Architectures and Compilers Network of Excellence members

Abstract

High-level loop optimizations are necessary to achieve good performance over a wide variety of processors. Their performance impact can be significant because they involve in-depth program transformations that aiming to sustain a balanced workload over the computational, storage, and communication resources of the target architecture. Therefore, it is mandatory that the compiler accurately models the target architecture and the effects of complex code restructuring.

However, because optimizing compilers (1) use simplistic performance models that abstract away many of the complexities of modern architectures, (2) rely on inaccurate dependence analysis, and (3) lack frameworks to express complex interactions of transformation sequences, they typically uncover only a fraction of the peak performance available on many applications. We propose a complete iterative framework to address these issues. We rely on the polyhedral model to construct and traverse a large and expressive search space. This space encompasses only legal, distinct versions resulting from the restructuring of any static control loop nest.

We first propose a feedback-driven iterative heuristic tailored to the search space properties of the polyhedral model. Though, it quickly converges to good solutions for small kernels, larger benchmarks containing higher dimensional spaces are more challenging and our heuristic misses opportunities for significant performance improvement. Thus, we introduce the use of a genetic algorithm with specialized operators that leverage the polyhedral representation of program dependences. We provide experimental evidence that the genetic algorithm effectively traverses huge optimization spaces, achieving good performance improvements on large loop nests.

1. Introduction

In recent years, feedback-directed iterative optimization has become a promising direction to harness the full potential of future and emerging processors with modern compilers. Building on operation research, statistical analysis and artificial intelligence, iterative optimization generalizes profile-directed approach to integrate precise feedback from the runtime behavior of the program into optimization algorithms. Through the many encouraging results that have been published in this area, it has become apparent that achieving better performance with iterative techniques depends on two major challenges.

Search space expressiveness. To achieve strong and portable performance with iterative techniques, it is essential that transformation search space be expressive enough to let the optimizations target all important architecture components and address all dominant performance anomalies,

Search space traversal. It is also important to construct search algorithms (analytical, statistical, empirical) and acceleration heuristics (performance models, machine learning) that effectively traverse the search space by exploiting its static and dynamic characteristics.

This paper targets the optimization problem of selecting an affine transformation to optimize imperfectly nested loops. Girbal et al. show that complex sequences of loop transformations are needed to generate efficient code for full-size loop nests on modern architectures [19]. They also show that such transformation sequences are out of reach of classical loop optimization frameworks, although multidimensional affine scheduling [18, 15] can successfully model them *as one single optimization step* [47], and scales to *large loop nests* with hundreds of array references [19]. Within this space of complex sequences of loop transformations, our work is the first to simultaneously address the two above-mentioned challenges.

We make the following contributions.

1. By considering multidimensional schedules, we tackle *any possible static-control loop nest* in a program, a major leap forward from the state-of-the-art iterative optimization on one-dimensional schedules [35]. We simultaneously extend the application domain — applicability and scalability to larger codes — and search space expressiveness — construction of more complex sequences of transformations.
2. To harness the combinatorial explosion of the optimization problem, we present a scalable traversal heuristic and original genetic operators, tailored to efficiently traverse a space of *legal*, *distinct* multidimensional schedules. These operators leverage the algebraic structure and statistical properties of the space.

3. We simultaneously demonstrate good performance gains and excellent convergence speed on huge search spaces, even on larger loop nests (up to 20 loops) where iterative affine scheduling has never been attempted before.
4. We demonstrate significant performance improvements on three different architectures.

The paper is structured as follows. Section 2 introduces multidimensional scheduling in the polyhedral model. Section 3 constructs the search space of legal, distinct versions (multidimensional schedules) for a program, and the key properties of this space. Section 4 proposes a first heuristic to efficiently traverse this space in the case of small kernels. Section 5 defines a genetic algorithm with specialized mutation and reproduction operators that can effectively traverse huge search spaces associated with larger loop nests. We show that our custom genetic algorithm achieves good performance improvements despite the poor statistical distribution of performance-enhancing schedules. Section 6 discusses related work, before we conclude in Section 7.

2. Thinking in Polyhedra

Most compilers' internal representations match the inductive semantics of imperative programs (syntax tree, call tree, control-flow graph, SSA). In such reduced representations of the dynamic execution trace, a statement of a high-level program occurs only once, even if it is executed many times (e.g., when enclosed within a loop). Representing a program this way is not convenient for aggressive optimizations which often need to consider a representation granularity at the level of dynamic *statement instances*. For example, complex transformations like loop interchange, fusion or tiling operate on the execution order of statement instances [48]. Due to compilation-time constraints and to the lack of an adequate algebraic representation of the semantics of loop nests, classical (non-iterative) compilers are unable to adapt the schedule of statement instances of a program to best exploit the architecture resources. For example, compilers can typically not apply any transformation if data dependences are non-uniform (unimodular transformations, tiling), if the loop trip counts differ (fusion) or simply because profitability is too unpredictable. As a simple illustration, consider the Ring-Roberts edge detection filter shown in Figure 1. While it is straightforward to detect a high level of data reuse between the two loop nests, none of the compilers we considered — Open64 4.0, ICC 10.0, PathScale 3.0, GCC 4.2.0 — were able to apply loop fusion for a potentially 50% cache miss reduction when arrays do not fit in the data cache (plus additional scalar promotion and instruction-level-parallelism improvements). Indeed, this apparently simple transformation actually requires a non-trivial composition of (two-dimensional) loop shifting, fusion and peeling.

```
  /* Ring blur filter */
  for (i=1;i<length-1;i++)
    for (j=1;j<width-1;j++)
R     Ring[i][j]=(Img[i-1][j-1]+Img[i-1][j]+Img[i-1][j+1]+
                  Img[i][j+1]  +            Img[i][j-1]   +
                  Img[i+1][j-1]+Img[i+1][j]+Img[i+1][j+1])/8;

  /* Roberts edge detection filter */
  for (i=1;i<length-2;i++)
    for (j=2;j<width-1;j++)
P     Img[i][j]=abs(Ring[i][j]-Ring[i+1][j-1])+
                abs(Ring[i+1][j]-Ring[i][j-1]);
```

Figure 1. `Ring-Roberts` edge detection for noisy images

To build complex loop transformations, a well known alternative is to represent programs in the *polyhedral model*. It is a flexible and expressive representation for loop nests with statically predictable control flow. The polyhedral model captures control-flow and data-flow with three linear algebraic structures, described in the following subsections. Such loop nests amenable to algebraic representation are called *static control parts* (SCoP) [16, 19].

2.1 Iteration Domains

Iteration domains capture the dynamic instances of all statements — all possible values of surrounding loop iterators — through a set of affine inequalities. For example, statement R in Figure 1 is executed for every value of the pair of surrounding loop counters, called the *iteration vector*: the iteration vector of statement R is $\vec{x}_R = (i, j)$. Hence, the iteration domain of R is defined by its enclosing loop bounds:

$$\mathcal{D}_R = \{i, j \mid 1 \le i \le \text{length}-1 \wedge 1 \le j \le \text{width}-1\},$$

which forms a parametric polyhedron (a space bounded by inequalities, a.k.a. *hyperplanes* or *faces*). Each integral point inside this polyhedron corresponds to exactly one execution of statement R. This model allows the compiler to manipulate statement execution and iteration ordering at the most precise level.

2.2 Subscript Functions

Subscript functions capture the data locations on which statement operate. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts of the form of affine expressions which may depend on surrounding loop counters (e.g., i and j for statement R) and global parameters (e.g., length and width). Each subscript function is linked to an array that represents a read or a write access. For instance, the subscript function for the read reference `Img[i-1][j]` of statement R is simply $f(i, j) = (i-1, j)$.

2.3 Multidimensional Schedules

Iteration domains define exactly the set of dynamic instances for each statement. However this algebraic structure does not describe the order in which each instance has to be executed with respect to each other. Of course, we do not want to rely on the inductive semantics of the sequence and loop iteration for this purpose, as it would break the algebraic reasoning about loop nests.

A convenient way to express the execution order is to give each instance an execution date. It is obviously impractical to define them one by one since instance number may be either very large or unknown at compile time. An appropriate solution is to do it at the statement level by defining, for each statement, a *scheduling* function that specifies the execution date for each instance of a corresponding statement. For tractability reasons, we restrict these functions to be affine (relaxation of this constraint may exists [6], but challenges the code generation step [7]).

This work deals with *multidimensional 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

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

where T_S is a matrix of constants (possibly not integers). Multidimensional dates can be seen as clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes, and so on. Unlike one-dimensional affine schedules, every static control program has a multidimensional affine schedule [18]. Hence the application domain of the present work extends to *all* static control parts in general programs.

For a concrete intuition of scheduling functions, lets go back to Ring-Roberts example of Figure 1. A possible multidimensional schedule is: $\theta_R(i,j) = (i,j)$ and $\theta_P(i,j) = (i + \text{length} - 2, j)$. This means the schedule of statement R orders its instances according to i first and then j. This matches the structure of the first loop nest. This is similar for statement P, except for the offset on the first time-dimension which states that the first nest runs before the second one: while the largest value of the first time-dimension for R is $\text{length} - 2$, the smallest value of the first dimension of P is $\text{length} - 1$. Hence the loop surrounding P "starts" after the loop surrounding R. Efficient algorithms and tools exist to generate a target code from a polyhedral representation with multidimensional affine schedules [38, 7]. Recent work by Girbal et al. [19] and Vasilache et al. [43] improved these algorithms to scale up to thousands of statements.

2.4 Benefits of a Polyhedral Representation

Reasoning about programs in a polyhedral representation has many advantages, for both program analysis and transformation.

1. exact dependence analysis is possible [16, 36, 30];
2. there exist efficient algorithms and tools to regenerate imperative code [38, 7];
3. loop transformation sequences of arbitrary complexity can be constructed and transparently applied in one single step.

A more complete description of static control parts was given by Xue [49] and their applicability to compute intensive, scientific or embedded applications have been extensively discussed by Girbal et al. [19, 32]. Frameworks to highlight SCoPs in general programs and to extract both iteration domains and subscript functions already exist or are in active development in compiler platforms like WRAP-IT/URUK for Open64 [19], and Graphite for GCC [33].

Multidimensional affine schedules support arbitrary complex compositions of a wide range of program transformations. Several frameworks have been designed to facilitate the expression of such transformations [22], or to enable their composition and semi-automatic construction [19, 44]. As illustration, a trivial loop fusion is not possible to improve data locality on the Ring-Roberts kernel in Figure 1. Because of both data dependences and non-matching loop bounds, only a partial loop fusion is possible, which translates into a sequence of, e.g., *fusion*, *shifting* and *index-set splitting* [48]. Using multidimensional schedules, a correct transformation (found using *chunking* [8]) is simply: $\theta_R(i,j) = (i,j)$ and $\theta_P(i,j) = (i + 2, j)$. One may care to check, using any polyhedral code generator, that the corresponding target code corresponds to a quite complex composition of syntactic transformations.

3. Generating Program Versions

The space of multidimensional affine schedules is very expressive. Each point in the space corresponds to potentially very different program versions, exposing a wide spectrum of interactions between architectural components and back-end compiler optimizations. This section presents the formal construction of the *space of legal, distinct schedules only*. We also give a practical heuristic to reduce the combinatorics of any algorithm to traverse this space while preserving the legality property.

3.1 Generating All Legal Schedules

Nisbet [31], then Long and Fursin [28] noticed that choosing a schedule at random likely to lead to an illegal program version, and that the probability of finding a *legal* one (which do not alter semantics) decreases exponentially with program size.

This challenge can only be tackled when integrating data dependence information into the construction of the search space. Two statement instances are in dependence if they access the same memory location and at least one of these accesses is a write. Maintaining the relative order of such instances is a sufficient condition to preserve the original program semantics [9].

Dependences in static control parts can be expressed by *dependence polyhedra* whose formal description has been proposed by Feautrier [16]. A dependence polyhedron $\mathcal{D}_{R,S}$ is a subset of the Cartesian product of the iteration domains of statements R and S. Each integral point of a dependence polyhedron corresponds to a pair of instances of the statements in dependence. Thus, a schedule does not change the semantics of the original program if it satisfies the *precedence constraint*: for all pairs of iteration vectors $\vec{x_R}$ and $\vec{x_S}$ in all dependence polyhedra,

$$\theta_R(\vec{x_R}) \prec \theta_S(\vec{x_S}),$$

where $\prec$ denotes the *lexicographic ordering*.[1]

The affine form of Farkas lemma allows to translate such constraints into an *affine* equivalent [40]. Feautrier used this result to express every constraint that a one-dimensional schedule must respect to preserve the semantics of the original program [17]. Those constraints bound a space where each integral point corresponds to a legal schedule. Pouchet et al. showed it is possible to traverse this space efficiently for small programs that accept one-dimensional schedules [35]. But dealing with multidimensional schedules leads to a combinatorial explosion.

Using one-dimensional schedules, all dependences have to be satisfied within a single time dimension: the precedence constraint is simply $\theta_R(\vec{x_R}) < \theta_S(\vec{x_S})$ and θ is a row vector. In multidimensional schedules, the legality constraints can also be built time dimension per time dimension, with the difference that a dependence needs to be *weakly solved* — $\theta_S(\vec{x_S}) - \theta_R(\vec{x_R}) \succeq 0$ — for the first time dimensions until it is *strongly solved* — $\theta_S(\vec{x_S}) - \theta_R(\vec{x_R}) > 0$ — at a given time dimension d. Once a dependence has been strongly solved, no additional constraint is required for legality at dimensions $d' > d$. Reciprocally, a dependence must be weakly solved for all $d'' < d$. There is freedom to *decide* at which time dimension a dependence will be strongly solved. Each possible decision leads to a potentially different search space. Furthermore, it is possible to arbitrarily increase the number of time dimensions of the schedule, resulting in an infinite set of scenarios in general.

The ouptut is in the form of a list of polyhedra of legal schedules, one for each time dimension.

A naive solution to build (a representative subset of) all multidimensional schedules would be to restrict the possible scenarios by setting an upper bound on the number of time dimensions (e.g., the loop nest depth + 1), which is already unrealistic for programs of more than a few dependences.

Note that loop tiling (a.k.a. blocking) is not directly expressible on multidimensional schedules. It requires modifications of the iteration domain (insertion of new dimensions) [2, 19] or specific handling in the code generator [39]. Because of this, our search space does not currently encompass loop tiling. Recent results by Renganarayanan et al. and Bondhugula et al. are promising directions towards fully integrating loop tiling with affine scheduling algorithms [39, 11].

3.2 Building a Practical Search Space

We have to face a double combinatorial problem. First, there are too many polytopes to be considered. For instance, the Ring-Roberts filter shown in Figure 1 has 12 dependence polyhedra, from which follows a huge number of possible strongly/weakly solved dependence scenarios. Second, one needs to limit the search to bounded

[1] $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ iff there exists an integer $1 \leq i \leq min(n,m)$ s.t. $(a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1})$ and $a_i < b_i$.

polytopes. Yet, even the smallest bound leads to polytopes that are too large to be explored exhaustively for complex loop nests.

Feautrier found a systematic solution to the explosion of the number of polyhedra: he considers a space of legal schedules leading to maximum fine-grain parallelism [18, 45]. To achieve this, a greedy algorithm maximizes the number of dependences solved for a given dimension. While this solution is interesting because it reduces the number of dimensions and exhibits inner parallelism, it is not practical enough for several reasons. First, it needs to solve a system of linear inequalities involving every schedule coefficient *plus* a decision variable per dependence [18]. This makes the problem untractable for all but small kernels. Moreover, minimizing the number of dimensions often translates into big schedule coefficients; these generally lead to algorithmic complexity and control overhead after generation of the target imperative code [22].

We suggest a simple variation to overcome those issues. The following algorithm sketches our search space construction for a given static control part:

```
1. Compute the exact set $G$ of dependences for the SCoP through
   instancewise analysis [16]
2. $d \leftarrow 1$
3. while $G \neq \emptyset$ do
   (a) Initialize $\mathcal{L}_d$ (the space of legal schedules for time dimension
       $d$) to the full-space polyhedron
   (b) for each dependence $\mathcal{D}_{R,S} \in G$
       • Compute $\mathcal{W}_{\mathcal{D}_{R,S}}$ — the space of legal schedules weakly
         satisfying only $\mathcal{D}_{R,S}$ — by enforcing, for all pairs of
         points in $\mathcal{D}_{R,S}$:
         $\theta_S(\vec{x_s}) - \theta_R(\vec{x_R}) \geq 0$
       • $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{W}_{\mathcal{D}_{R,S}}$
   (c) for each dependence $\mathcal{D}_{R,S} \in G$
       • Compute $\mathcal{S}_{\mathcal{D}_{R,S}}$ — the space of legal schedules strongly
         satisfying only $\mathcal{D}_{R,S}$ — by enforcing, for all pairs of
         points in $\mathcal{D}_{R,S}$:
         $\theta_S(\vec{x_s}) - \theta_R(\vec{x_R}) > 0$
       • if $\mathcal{L}_d \cap \mathcal{S}_{\mathcal{D}_{R,S}} \neq \emptyset$ then
           ▪ $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{S}_{\mathcal{D}_{R,S}}$
           ▪ $G \leftarrow G - \mathcal{D}_{R,S}$
   (d) $d \leftarrow d + 1$
```

This heuristic outputs for each schedule dimension d a space $\mathcal{L}_d$ of legal solutions.

The algorithm terminates; the proof uses the same argument as Feautrier's muldidimensional scheduling algorithm [45]: at least one dependence can be strongly solved per time dimension d. Nevertheless, it differs from Feautrier's algorithm as it does not guarantee a maximal number of dependences solved per dimension. Therefore it may not minimize the number of dimensions of the schedule: this is not an issue as we only consider sequential codes.[2] However, this algorithm is efficient and only needs one polyhedron emptiness test per dependence,[3] and the elimination of Farkas multipliers used to enforce the precedence constraint on schedule coefficients is performed dependence per dependence (i.e., on very small systems) [35]. Since we consider sequential codes only, we can bound the coefficient values within $\{-1, 0, 1\}$ to minimize control-flow overhead. This would have been very restrictive if we were constrained to one-dimensional schedules. In the multidimensional case, although it eliminates some schedules from the space (e.g., non-unit skewing), these bounds are compatible with the expression of arbitrary compositions of loop fusion, distribution, interchange, code motion; in the worst case, it translates into additional time dimensions. Overall, this solution gives an interesting tradeoff between scalability and expressiveness (performance of the generated code).

So far, we did not define the order in which dependences are considered when checking against strong satisfaction. This order can have a dramatic impact on the constructed space. A long term approach would be to consider this order as part of the search space, but this is not currently practical (combinatorial explosion). Instead, we use two analytical criteria to order the dependences. First of all, each dependence is assigned a priority, depending on the memory traffic generated by the pair of statements in dependence. We use a simplified version of the model by Bastoul and Feautrier [8]: for each array A and dimension d, we approximate the traffic as $m_d^{r_A}$, where m_d is the size of the d^{th} dimension of the array, and r_A is the rank of the concatenation of the subscript matrices of all references to dimension d of array A in the statement. Thus the generated traffic evaluation for a given statement is a multivariate polynomial in the parametric sizes of all arrays. We use profiling to instantiate these size parameters. Intuitively, maximizing the depth where a dependence is strongly solved maximizes reuse in inner loops and minimizes the memory traffic in outer loops. Therefore, we start with dependences involved in the statements with the less traffic. Our second criterion is based on *dependence interference*; it is used in case of non-discriminating priorities resulting from the first criterion. Two dependences interfere if it is impossible to build a one-dimensional schedule strongly satisfying these two dependences. We first try to solve dependences interfering with the lower number of other dependences, maximizing our chance to strongly solve more dependences within the current time dimension.

3.3 Scanning the Search Space Polytopes

The algorithm presented in Section 3.2 constructs one polytope per dimension of the schedule. Picking one point in every polytope fully describes one multidimensional schedule, hence one program version: the generated imperative codes will be distinct if the scheduling matrices are distinct. This is reminiscent of the classical polyhedron scanning problem [4, 38, 7]; however, none of the existing algorithms scale to the hundreds of dimensions we are considering. Fortunately, our problem happens to be simpler than "static" loop nest generation: we only need to "dynamically" enumerate every integral point which respects the set of constraints.

Each program version is represented by a unique scheduling matrix Θ. The first columns are schedule coefficients associated with each loop iterator surrounding a statement in the original program, for all statements ($\vec{\imath}$). The next set of columns are schedule coefficients associated with global parameters ($\vec{p}$), for all statements. The last column are the schedule coefficients associated with the constant (c), for all statements.

Since we represent legal schedules as multidimensional affine functions, each row Θ_d of the scheduling function corresponds to an integer point in the polytope of legal coefficients $\mathcal{L}_d$, built explicitly for this dimension. A program version in the optimization space can thus be represented as follows, for a SCoP of l statements, a schedule of dimension s, and the iteration vector $\vec{x}$:

[2] Affine partitioning may be better suited to characterize parallelism in the polyhedral model [27].

[3] Over $\mathcal{L}_d$ which contains exactly one variable per schedule coefficient.

$$\Theta.\vec{x} = \begin{pmatrix} \vec{\imath}_1^1 & \cdots & \vec{\imath}_p^1 & \vec{p}_1^1 & \cdots & \vec{p}_p^1 & c_1^1 & \cdots & c_p^1 \\ \vdots & & & & & & & & \vdots \\ \vec{\imath}_1^s & \cdots & \vec{\imath}_p^s & \vec{p}_1^s & \cdots & \vec{p}_p^s & c_1^s & \cdots & c_p^s \end{pmatrix} \cdot \begin{pmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_p \\ \vec{n}_1 \\ \vdots \\ \vec{n}_p \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

To build each row Θ_d, we scan the legal polytope $\mathcal{L}_d$, by successively instantiating values for each coefficient in a predefined order.[4] Fourier-Motzkin elimination — a.k.a. projection — [5] provides a representation of the affine constraints of a polytope fitted for its dynamic traversal. Computing the projection of all variables of a polytope $\mathcal{L}_d$ results in an equivalent polytope where it is guaranteed that the value of v_k is a *function* of $v_1, \dots, v_{k-1}$, with affine inequalities involving only $v_1, \dots, v_k$. Thus, the sequential order to build coefficients is simply the reverse order of the elimination steps. This scheme guarantees that provided a value in the projection of $v_1, \dots, v_{k-1}$, a value exists for v_k, for all k.[5] In order to achieve scalability, we use a modified and redundancy-aware version of the Fourier-Motzkin algorithm. In its basic form, the algorithm is known to generate many redundant constraints; these redundancies reduce its scalability on large polyhedra. We improved it, while maintaining the following properties for each variable elimination:

1. any constraint defining a hyperplane parallel to an existing constraint is removed (this is trivially computed since the constraints are kept normalized);
2. any variable which is linearly dependent to any other one(s) is removed (thanks to implicit equalities detection and Gaussian elimination);
3. constraints are removed if, once opposed, no point exists in the solution polytope (we apply the Le Fur descending method [25]).

In practice, this modified algorithm scales to hundreds of variables (schedule coefficients) in the original system.

3.4 Schedule Completion Algorithm

For SCoPs with more than 4 or 5 statements, the previous construction leads to very large search spaces, challenging any traversal algorithm. It is possible to focus the search on some coefficients of the schedule with maximal impact on performance, postponing the instantiation of a full schedule in a second heuristic step. We show that such a two-step procedure can be designed without breaking the fundamental legality property of the search space. This approach will be used extensively to simplify the optimization problem.

The previous projection pass guarantees it is possible to complete — or even correct — any vector, slightly modifying its coefficients to make it lie within a given polytope. We use this property to design the following *completion algorithm*. Given a vector $\vec{v}$ of size n, for $k \in [1, n]$:

1. compute the lower bound and upper bound of v_k, provided the coefficient values for $v_1 \dots v_{k-1}$, and
2. if $v_k \notin [lb, ub]$, then $v_k = lb$ if $v_k < lb$ or $v_k = ub$ if $v_k > ub$.[6]

Therefore it is possible to partially build a schedule prefix, e.g., values for the $\vec{\imath}$ coefficients, while setting all other coefficients to 0. Then, applying this correction principle will result in finding the minimal amount of complementary transformations to make the transformation lie in the computed legal space. The completion algorithm motivates the order of coefficients in the Θ matrix. We showed that the most performance impacting transformations (interchange, skewing, reversal) are embedded in the first coefficients of Θ — the $\vec{\imath}$ coefficients; followed by coefficients usually involved in fusion and distribution — the $\vec{p}$ coefficients; and finally the less impacting c coefficients, representing loop shifting and peeling [34]. The completion algorithm finds complementary transformations in order of least to most impacting, as it will not alter any vector prefix if a legal vector suffix exists in the space.

Three fundamental properties are embedded in this completion algorithm:

1. if $v_1, \dots, v_k$ is a prefix of a legal point v, a completion is always found;
2. this completion will only update $v_{k+1}, \dots, v_{d_{\max}}$, if needed;
3. when $v_1, \dots, v_k$ are the $\vec{\imath}$ coefficients, the heuristic looks for the smallest absolute value for the $\vec{p}$ and constant coefficients, which corresponds to maximal (nested) loop fusion — relative to the $\vec{\imath}$ coefficients.

Picking coefficients as close as possible to 0 has several advantages in general: smaller coefficients tend to simplify code generation, improve locality, reduce latency, and increase the size of basic blocks in inner loops.

4. Traversing the search space

While it is possible to exhaustively traverse the constructed space of legal versions for little SCoPs, in the case of one-dimensional schedules, it becomes unpractical in the multidimensional case. Pouchet et al. give a preliminary answer by means of a heuristic to narrow this space and accelerate the traversal [35]. We build on this result to design a powerful heuristic suitable for the multidimensional case.

4.1 A Multidimensional Decoupling Heuristic

Our approach is called the *decoupling heuristic* as it leverages the completion algorithm of Section 3.4 to stage the exploration of large search spaces. It derives from the observation of the performance distribution, where density patterns hinted that not all schedule coefficients have a significant impact on performance [35, 34]. The principle of the decoupling heuristic for one-dimensional schedules is (1) to enumerate different values for the $\vec{\imath}$ coefficients, (2) to instantiate full schedules with the completion algorithm, and (3) to select the best completed schedules and further enumerate the different coefficients for the $\vec{p}$ part.

A direct extension to the multidimensional case exhibits two major drawbacks. First, the relative performance impact of the different schedule dimensions must be quantified. Second, an exhaustive enumeration of $\vec{\imath}$ coefficients for all dimensions is out of reach, as the number of points exponentially increases with the number of dimensions. Figure 2 illustrates this assertion by summarizing the size of the legal polytopes for different benchmarks, for all schedule dimensions. We consider 10 SCoPs extracted from classical benchmarks. The first eight are UTDSP benchmarks [26] directly amenable to polyhedral representation: `compress-dct` is an image compression kernel (8x8 discrete cosine transform),

[4] The order has no impact on the completeness of the traversal.

[5] The case of holes in $\mathbb{Z}$-polyhedra is handled through a schedule completion algorithm described in the next section.

[6] $\mathbb{Z}$-holes are detected by checking if $lb > ub$.

`edge-convolve2d` is an edge detection kernel (different from Ring-Roberts), `fir` is a Finite Impulse Response filter, `lmsfir` is a Least Mean Square adaptive FIR filter, `iir` is an Infinite Impulse Response filter, `matmult` is a matrix multiplication kernel, `latnrm` is a normalized lattice filter, and `lpc (LPC_analysis)` is the hot function of a linear predictive coding encoder. We considered two additional benchmarks: `ludcmp` solves simultaneous linear equations by LU decomposition, and `radar` is a real code for the analysis of radar pulses. For each benchmark, we report the number of (complex) instructions carrying array accesses (#Inst), the number of loops (#Loops), dependences (#Dep), schedule dimension (#Dim), and the total number of points for those dimensions (still only legal schedules).

Relations between schedule dimensions To extend the decoupling approach to multidimensional schedules, we need to integrate interactions between dimensions. For instance, to distribute the outer loop of a nest (which can improve locality and vectorization [3]), one can operate on the $\vec{p}$ and c parts of the schedule for the first dimension (a parametric shift). On the other hand, altering the $\vec{\imath}$ parts will lead to the most significant changes in the loop controls. Indeed, the largest performance variation is usually captured through the $\vec{\imath}$ parts [34], and a careful selection of those coefficients is mandatory to attain the best performance; conversely, it is likely that the best performing transformations will share similar $\vec{\imath}$ coefficients in their schedules.

Furthermore, the first dimension is highly constrained in general, since all dependences need to be — weakly or strongly — considered. Conversely, the last dimension is the less constrained and often carries only very few dependences.[7]

The decoupling heuristic in a nutshell We conducted an extensive experiment showing that Θ_1 (the first time dimension of the schedule) is a major discriminant of the overall performance distribution [34]. Therefore, the heuristic starts with an exploration of values for coefficients of Θ_1, completing the schedule with a single value for the remaining time dimensions (that is, the rest of the schedule is set to 0 and the completion algorithm is called on it). Like the decoupling heuristic, this exploration is limited to the subspace associated with the $\vec{\imath}$ coefficients of Θ_1, except if this subspace is smaller than a given constant L_1 — $L_1 = 50$ in our experiments. L_1 drives the exhaustiveness of the procedure: the larger the degree of freedom, the slower the convergence. By limiting to the $\vec{\imath}$ class we target only the most performance impacting subspaces.

To enumerate points in the polytopes, we incrementally pick a dimension then *pick an integer* in the polyhedron's projection onto this dimension. Note that the full projection is computed once and for all by the Fourier-Motzkin algorithm presented in Section 3.3, *before* traversal. Technically, to enumerate integer points of the subspace composed of the first m columns of $\mathcal{L}_d$, we define the following recursive procedure to build a points $\vec{v}$:

```
EXPLORE ($\vec{v}, k$) :
  1. compute the lower bound and upper bound of $v_k$, provided the
     coefficient values for $v_1 \dots v_{k-1}$;
  2. for each $x \in [lb, ub]$, set $v_k = x$;
     if $k < m$ call EXPLORE ($\vec{v}, k+1$) else output $\vec{v}$.
```

The enumeration is initialized with a call to EXPLORE $(\vec{v}, 1)$. The completion algorithm is then called on each vector $\vec{v}$ generated, to compute a legal suffix for $\vec{v}$ (corresponding to the columns $[m+1, n]$ of $\mathcal{L}_d$), finally instanciating a legal point of full dimensionality.

Then, the heuristic selects the $x\%$ best values for Θ_1 ($x = 5\%$ in our experiments), it proceeds with the exploration of values for coefficients of Θ_2, and recursively until the last but one dimension of the schedule. The last dimension (corresponding to the innermost nesting depth in the generated code) is not traversed, but completed with a single value: exploring it would yield a huge number of iterations, with limited impact on the generated code, and negligible impact on performance. Eventually, the exploration is bound with a static limit — 1000 evaluations in our experiments.

4.2 Experiments

We consider three target architectures. The AMD Alchemy Au1500 is an embedded SoC with a MIPS32 core (Au1) running at 500MHz. We used GCC 3.2.1 with the -O3 flag (version of GCC and option with peak performance numbers, according to the manufacturer). The STMicroelectronics ST231 is an embedded SoC with a 4-issue VLIW core running at 400MHz and a blocking cache. We used st200cc 1.9.0B (Open64) with the flags `-O3 -mauto-prefetch -OPT:restrict`. The AMD Athlon X64 3700+ has a 1MB L2 cache and runs at 2.4GHz. It runs Mandriva Linux and the native compiler is GCC 4.1.1. We used the following optimization settings for this platform which are known to bring excellent performance: `-O3 -msse2 -ftree-vectorize` . For this particular machine, hardware counters were used to collect fine-grained cycle counts, and we used a real-time priority scheduler to minimize OS interference. We picked the average of 10 runs for all performance evaluations.

We implemented an instancewise dependence analysis, the construction of the space of legal transformations, and the efficient scanning algorithms introduced in this paper.[8] We used free software such as `PipLib` [16, 46] (a polyhedral library and parametric integer linear programming solver) and `CLooG` [7] (an efficient code generator for the polyhedral model). For each point in the search space, (1) we generated the kernel C code with `CLooG`,[9] (2) then we integrated this kernel in the original benchmark along with instrumentation to measure running time (we use performance counters when available), (3) we compiled this code with the native compiler and appropriate options, (4) and finally run the program on the target architecture and gather performance results. The original code is included in this procedure starting at the second step, for appropriate performance comparison. Finally, the full iterative compilation and execution process takes a few seconds on the heuristics, and up to a few minutes on the GA described Section 5 for the largest benchmark (up to 1000 tested versions). The time to compute the legal space and to generate points is negligible with respect to the total running time of the tested versions.

Results Figure 3 shows the results for the three architectures we considered. We report the total numbers of tested versions (*Tested*), the run index of the best performing version (*Id Best*; the lower, the earlier), and the performance improvement in percentage (*Perf. Imp.*). We also imposed a static limit of evaluating 1000 data points in the search space.[10]

All UTDSP experiments use the reference parameters. Increasing data size would emphasize locality effects, yielding higher speedups. E.g., `matmult` on Athlon with $n = 250$ yields 3.61 speedup, $n = 64$ yields 3.18 speedup, whereas the reference value $n = 10$ yields 1.43% speedup.

Discussion Our results show significant improvement on all kernels of the UTDSP suite. In addition, about 50 runs were sufficient for kernels with less than 10 statements (all but `lpc` and `radar`).

[7] This is typically the case when the final dimension is required to order the statements within an innermost loop.

[8] LETSEE, the LEgal Transformation SpacE Explorator, beta version available at http://www-rocq.inria.fr/~pouchet/software/letsee

[9] We use CLooG version 0.14.0 with default options.

[10] It matches the maximum number of versions considered by the genetic algorithm in Section 5.

Benchmark	#Inst.	#Loops	#Dep.	#Dim.	dim 1	dim 2	dim 3	dim 4	Total
`compress-dct`	6	6	56	3	20	136	10857025	n/a	2.9×10^{10}
`edge`	3	4	30	4	27	54	90534	43046721	5.6×10^{15}
`iir`	8	2	66	3	18	6984	$> 10^{15}$	n/a	$> 10^{19}$
`fir`	4	2	36	2	18	52953	n/a	n/a	9.5×10^{7}
`lmsfir`	9	3	112	2	27	10534223	n/a	n/a	2.8×10^{8}
`matmult`	2	3	7	1	912	n/a	n/a	n/a	912
`latnrm`	11	3	75	3	9	1896502	$> 10^{15}$	n/a	$> 10^{22}$
`lpc`	12	7	85	2	63594	$> 10^{20}$	n/a	n/a	$> 10^{25}$
`ludcmp`	14	10	187	3	36	$> 10^{20}$	$> 10^{25}$	n/a	$> 10^{46}$
`radar`	17	20	153	3	400	$> 10^{20}$	$> 10^{25}$	n/a	$> 10^{48}$

Figure 2. Search space statistics

		compress-dct	edge	iir	fir	lmsfir	matmult	latnrm	lpc	ludcmp	radar	Average
AMD Athlon	#Runs	480	243	1000	77	1000	81	1000	1000	1000	1000	
	Id. Best	**19**	**11**	**34**	**33**	**51**	**16**	6	489	37	405	
	Perf. Imp.	37.11%	5.58%	37.50%	40.24%	30.98%	42.87%	15.11%	31.15%	4.50%	6.42%	25.14%
ST231	#Runs	480	243	1000	77	1000	81	1000	1000	1000	1000	
	Id. Best	**39**	**12**	**6**	**2**	**9**	**16**	13	158	391	709	
	Perf. Imp.	15.11%	3.10%	24.91%	17.96%	10.17%	17.91%	2.61%	1.99%	6.33%	4.12%	10.42%
Au1500	#Runs	480	243	1000	77	1000	81	1000	1000	1000	1000	
	Id Best	**30**	**17**	**38**	**27**	**11**	**17**	43	82	175	454	
	Perf. Imp.	22.37%	2.51%	3.12%	14.00%	15.80%	20.18%	15.19%	14.08%	3.66%	3.39%	11.43%

Figure 3. Results of the decoupling heuristic for AMD Athlon, ST231 and Au1500

For all benchmarks, the best program version is syntactically very far from the original one.

A good illustration of this is given for the `Ring-Roberts` running example, which achieve a 1.47 speedup on a full HD image on AMD Athlon; hardware counter details show a 54% reduction of the L1 hit/miss ratio and a 51% of the data TLB misses. This complex transformation is the result of multidimensional shifting and peeling of the iterations preventing from fusion, and the complete fusion of the remaining iterations.

The limited performance improvement for `edge-convolve2d` is directly correlated to the code structure: this benchmark performs a convolution of a 3x3 kernel, and is an excellent candidate for optimization with loop unrolling — a transformation not embedded in our search space. Our technique is fully compatible with other iterative search techniques such as parameters tuning [1], and it is expected that this combination would bring excellent performance in this case.

We also noticed that performance improvements are often the result of indirect enabling of back-end compiler optimizations (e.g., vectorization or scalar promotion), in addition to the direct impact on hardware components (e.g, locality). Modern compiler optimization heuristics are still very fragile, and the interactions between optimization phases are not captured in their design. Predicting this interaction on non-trivial codes is still out of reach, and slight syntactic differences can trigger different optimization results. Testing different source code having the same semantics is one way to circumvent the compiler's optimization unpredictability.

In addition, the best iteratively found transformation for a given benchmark is different when considering a different target architecture. This is due to different interactions with the compiler, as well as different architectural features to optimize for. Note that it is not a consequence of working with more expressive schedules: we already highlighted a similar pattern for the case of one-dimensional schedules [35]. It confirms the complexity of the optimization problem and the relevance of a feedback-directed approach.

The heuristic heavily relies on the observation that the first dimension of the schedule contains very few points — it traverses this dimension exhaustively. However, exhaustive enumeration is only possible for small kernels, such as most UTDSP benchmarks. Unfortunately, for larger programs like `lpc`, `ludcmp`, `radar`, and to some extent on `latnrm`, this approach does not scale.

To address this scalability issue, we substitute the exhaustive search with a genetic algorithm.

5. Evolutionary Traversal of the Polytope

This section introduces novel genetic operators tailored to the traversal of polytopes of legal affine schedules.

Genetic algorithms (GA) [20] are known for their genericity: we chose an evolutionary approach because of the natural encoding of the geometric properties of the search space into crossover and mutation operators. The two main properties are the following:

1. to enforce legality and uniqueness of the program versions, the search space polytope must be closed for the genetic operators; we construct dedicated mutation and crossover operators satisfying this property;
2. unlike random search, the traversal is characterized by its non-uniformity (from the initial population and the crossovers); this is utterly important as the largest part of the search space is generally plagued with poor or similar performing versions [35, 34].

Genetic algorithms have often be used in program optimization. Our contribution is to reconcile fine-grain control of a transformation heuristics — as opposed to optimization flag or pass selection [42, 1] — with the guaranteed legality of the transformed program — as opposed to filtering approaches [31, 29, 28] or always-correct transformations [41, 24].

5.1 Genetic Algorithm

Using classical GA operators would not be an efficient way to generate data points in our search space. This is because legal schedules lie in affine bounds that are strongly constrained and changing them at random has a very low probability of preserving legality. Moreover, in general, this probability decreases exponentially with the space dimension [31]. We thus need to understand the properties of the space of legal schedules, and to embed them into dedicated GA operators.

Some properties of affine schedules The construction algorithm outputs one polytope per schedule dimension. We can deduce numerous properties on these polytopes, either deriving from the construction algorithm or from affine scheduling itself. In the following, the term *affine constraint* refers to any dependence, iteration domain, or search bound constraint on coefficients of the schedule — the columns of Θ.

1. No affine constraint involves coefficients from different rows of Θ, since those coefficients are computed from distinct polytopes. Of course, multiple coefficients inside a row can be involved in a constraint.
2. Multiple coefficients involved in a constraint are called *dependent*. Each row can be partitioned into *classes of dependent coefficients*, where no constraint involves coefficients from different classes. E.g., in the polyhedron defined by $\{x_1 + x_2 \geq 0 \wedge x_3 \geq 0\}$ we say that the set $\{x_1, x_2\}$ is independent from the set $\{x_3\}$. Legality preservation is local to each class of dependent coefficients.

We design novel genetic operators exploiting and preserving these properties.

Initialization We first introduce an individual with a statically computed schedule, computed by applying the completion algorithm on all schedule coefficents (which were previously all set to 0). This choice shares its motivation with the decoupling heuristic in Section 4.1.

The rest of the population is initialized by performing aggressive mutations on this static schedule; we generate 30 to 100 individuals, depending on the space dimension. The initial population is heavily biased towards a particular subspace (typically the subspace of the $\vec{\imath}$ coefficients), emphasizing the non-uniformity of the traversal.

Mutation The mutation operator starts with the computation of the distribution of probabilities to alter every coefficient. This probability is driven by three factors; the first one derives directly from the heuristic of the one-dimensional case [35]:

- coefficients of the iteration vectors have a dramatic impact on the structure of the generated code; minor modifications trigger wild jumps in the search space;
- conversely, coefficients with little linear dependences with others may require ample mutations to trigger significant changes.
- the schedule dimension considered: lower dimensions and especially the scalar ones usually have a lower impact on performance.

In addition, we weigh the probabilities with a uniform annealing factor, to tune the aggressiveness of the mutation operator along with the maturation of the population.

We randomly pick a value *within the legal bounds for this coefficient*, and according to the distribution of probabilities. As this mutation may cause other coefficients to become incorrect, we then update the schedule with the completion algorithm depicted in Section 3.4; it is a simple update because the schedule prefix can be kept in the legal space, computing mutated coefficients in the reverse order of Fourier-Motzkin elimination.

We also experimented a simpler mutation operator, where the bounds to pick mutated values where not adjusted to the corresponding polytope of legal versions, applying our correction mechanism a posteriori. This approach did not prove very effective as coefficients are often correlated or severely constrained: randomly picking values for multiple correlated coefficients often leads to identical schedules after correction. Only an incremental application of the correction mechanism avoids the generation of many duplicates (which strongly degrade the effectiveness of the mutation operator).

Crossover We provide two crossover operators. The *row* crossover is dedicated to compensating the row-wise aspect of the mutation operator. Given two individuals represented by Θ and Θ', the row crossover operator randomly picks rows of either Θ or Θ' to build a new individual Θ''. This operator obviously preserves legality since there are no dependences between rows. Since the mutation operates within a schedule dimension, it may succeed in finding good candidates for a given row of Θ or Θ', but may mix these with ineffective rows. Combining these rows may lead to a good schedule, with a much higher probability than with mutation alone.

The *column* crossover, is dedicated to crossing independent classes of schedule coefficients (represented by sets of columns not connected by any affine constraint); this operator is quite original and specific to the geometrical properties of the search space. It can be seen as a finer-grained crossover operator. From two individuals Θ and Θ', it randomly selects an independent class from either parent — at every dimension — to build Θ''. When there is only one independent class in a given schedule dimension this operator behaves like the row crossover. We rely on the geometric properties of the polytope to compute linearly independent sets of variables for a given schedule dimension. These sets are computed once and for all, immediately after the search space polytope is built. This operator preserves legality as it only modifies independent sets of schedule coefficients.

Dependences constrain schedule coefficients in pairs of statements. Several transitive steps are needed to characterize all correlations between coefficients in a dependent class. This operator carefully refines the grain of schedule transformations, while preserving legality as it only modifies independent sets of coefficients.

Selection The selection process is a classical *numerus clausus* filtering, keeping half of the current population for the next generation. A better option would be to combine multiple metrics, including performance predictors (to avoid running the code) or hardware counters. We are currently investigating such techniques.

5.2 Experimental Results

Figure 4 summarize the results of the genetic algorithm applied to all benchmarks for all architectures. Row Heuristic/GA shows the fraction of the speedup achieved by the decoupling heuristic w.r.t. the genetic algorithm, and fractions are averaged for the benchmarks of less than 10 statements versus more than 10. We initialized the population with 30 to 100 individuals, and performed at most 10 generations; therefore, the maximum number of runs for each program was 1000.

Comparing these results with the table in Figure 3 shows the efficiency and scalability of our method. The genetic algorithm achieves strong speedups for the larger kernels; these speedups are much stronger than those of the decoupling heuristic for the larger benchmarks. On the other hand, the decoupling heuristic exposes 78–100% of the speedup obtained with genetic algorithms within the first 50 runs, for all kernels of less than 10 statements

Architecture	compress-dct	edge	iir	fir	lmsfir	matmult	latnrm	lpc	ludcmp	radar	Average
AMD Athlon	44.17%	7.86%	32.18%	40.70%	24.23%	42.87%	28.23%	45.84%	69.63%	40.18%	37.58%
Heuristic/GA	84.31%	82.26%	93.58%	98.86%	80.71%	100%	**53.57%**	**67.68%**	**6.52%**	**16.05%**	89.95% / **35.95%**
ST231	18.42%	3.29%	27.40%	18.81%	8.63%	17.91%	0.86%	3.44%	5.96%	28.32%	13.30%
Heuristic/GA	83.33%	94.52%	90.91%	95.48%	85.14%	100%	**92.30%**	**32.20%**	**22.26%**	**30.82%**	91.56% / **44.39%**
AMD Au1500	25.11%	3.03%	4.07%	14.10%	19.18%	22.67%	27.01%	17.43%	15.71%	30.87%	17.91%
Heuristic/GA	89.21%	82.83%	78.29%	99.50%	81.64%	88.93%	**55.55%**	**82.35%**	**16.56%**	**10.91%**	86.73% / **41.35%**

Figure 4. Results of the genetic algorithm. The Decoupling heuristics succeeds in discovering 78-100% of the speedup achieved by GA for all benchmarks of less than 10 statements. For larger benchmarks, the GA performs 2.46× better in average, and up to 16× better.

Results are better on AMD Athlon than on embedded processors, probably because the architecture is more complex: a good interaction between architectural components is harder to achieve and brings higher improvements. Conversely, the ST231 and AMD Au1500 have a predictable behaviour, more effectively harnessed by the back-end compiler, and showing less room for improvement; yet our results are still significant for such targets.

We report a detailed study of the representative `compress-dct` benchmark, on AMD Athlon. Figure 5 summarizes the results, and confirms the huge advantage of the GA given the statistically sparse and chaotic occurrence of performance-enhancing schedules. The first graph shows the convergence of our GA approach versus a Random traversal in the space of legal schedules (only legal points are drawn). The GA algorithm ran for 10 generations from an initial population of 50 individuals. Both plots are an average of 100 complete runs. On the second graph, we report the performance distribution of the legal space. We exhaustively enumerated and evaluated all points with a distinct value for the $\vec{\imath}+\vec{p}$ coefficients of the first schedule dimension, combined with all points with a distinct $\vec{\imath}$ value for the second one; a total of 1.29×10^6 schedules are evaluated. For each distinct value of the first schedule dimension (plotted in the horizontal axis), we report the performance of the Best schedule, the Worst one, and the Average for all tested values of the second schedule dimension. The third graph shows the performance distribution for all tested points of the second schedule dimension, provided a single value for the first one, sorted from the best performing one to the worse (the best performing schedule belongs to this chart). The difficulty to reach the best points in the search space is emphasized by their extremely low proportion: only 0.14% of points achieve at least 80% of the maximal speedup, while only 0.02% achieve 95% and more. Conversely, 61.11% degrade performance of the original code, while in total 10.88% degrade the performance by a factor 2.

Figure 5(a) shows that our GA converges much faster than random search: in 500 runs, the random search is only able to discover a 18% performance improvement; the GA takes only 120 runs to match this figure, converging towards the space maximal 44.1% improvement after 350 runs, at the 7th generation (i.e., before the imposed limit of 10 generations). This the maximum speedup available, as shown by the exhaustive search experiments in Figure 5(b). The effectiveness of the genetic operators is illustrated by the decorrelation of the performance improvements and the actual performance distribution. Conversely, random traversal follows the shape of the performance distribution, and in average is not able to reach the best performing schedules — as their density in the space is very low.

Finally, we studied the behavior of multiple schedules for the `compress-dct` benchmark, analyzing hardware counters on Athlon. This study highlights complex interactions between the memory hierarchy (both L1 and L2 accesses must be minimized to achieve good performance), vectorization, and the activity of functional units. The best performing transformation reduces the numbers of stall cycles by a factor of 3, while improving the L2 hit/miss ratio by 10%. Transformation sequences achieving the optimal performance are opaque at first glance: they involve complex combinations of skewing, reversal, distribution and index-set splitting. These transformations address specific performance anomalies of the loop nest, but they are often associated with the interplay of multiple architecture components. Moreover, we observe that the best optimizations are usually associated with more complex control flow than the original code. The number of dynamic branches is increased in most cases [34], although stall cycles are heavily reduced due to locality and ILP improvements.

Overall, our results confirms the potential of iterative optimization to accurately capture the complex behavior of the processor and back-end compiler, and extends its applicability to optimization problems far more complex than those commonly solved in adaptive compilation.

6. Related Work

In recent years, the benefits of iterative compilation have been widely reported [23, 13, 14, 21]. Iterative compilation is often able to find optimization sequences that out-perform the highest optimization settings in commercial compilers.

Kulkarni *et al.* [24] introduce the VISTA system, an interactive compilation system which concentrates on reducing the time to find good solutions. Another system that attempted to speedup iterative compilation was introduced by Cooper *et al.* called ACME [12]. Triantafyllis *et al.* [42] develop an alternative approach to reduce the total number of evaluations of a new program. Here the space of compiler options is examined off-line on a per function basis and the best performing ones are classified into a small tree of compiler options.

Iterative optimization has been used effectively on a variety of compilation and parallelization problems and its applicability and practicality has been demonstrated beyond the academic world [37]. Although multidimensional affine scheduling is an obvious target for iterative optimization, its profitability is one of the most difficult to assess, due to (1) the model's intrinsic expressiveness (the downside of its effectiveness) and (2) its lack of analytical models for the impact of transformations on the target architecture. Hence, related work has been very limited up to this point. To the best of our knowledge, Nisbet pioneered research in the area with one of the very first papers in iterative optimization. He developed the GAPS framework [31] which used a genetic algorithm to traverse a search space of affine schedules for automatic parallelization. In addition, Long and O'Boyle [29] considered a larger search space of transformation sequences represented as multidimensional affine schedules. Both of these approaches suffer from under-constraining the search space by considering all possible schedules, including illegal ones. Downstream filtering approaches do not scale, due to the exponentially diminishing proportion of legal schedules with respect to the program size. For instance, Nisbet obtains only $3-5\%$ of legal schedules for the ADI benchmark (6 statements). Moreover, under-constraining the search space lim-

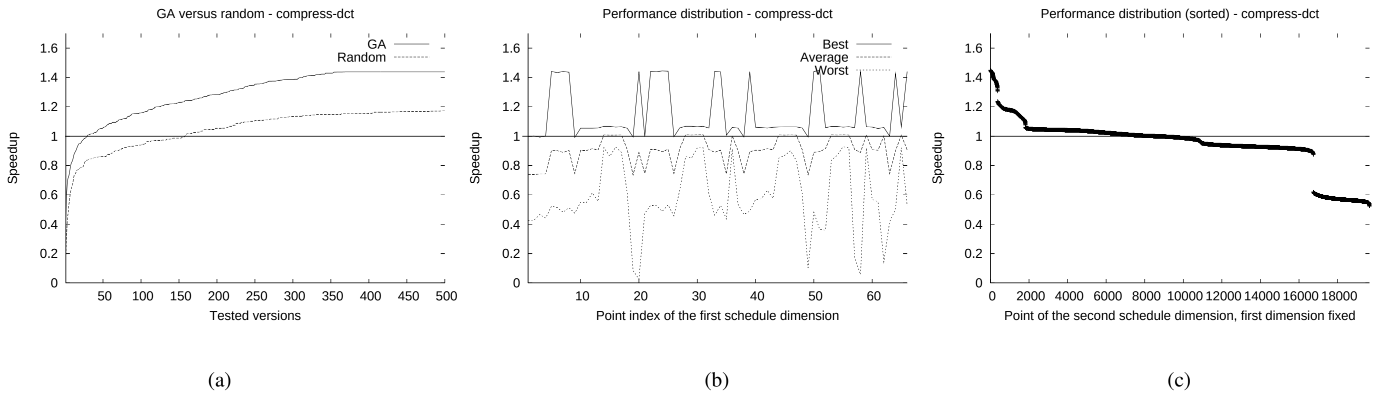

Figure 5. Performance Distribution of `compress-dct`, AMD Athlon. GA discovers the maximum speedup available in the search space.

its the possibility to narrow the search to the most promising subspaces.

Pouchet et al. demonstrated a more efficient approach, by embedding program dependences and affine scheduling properties into the search space itself. However, this approach was only applied to small kernels with single dimensional schedules. We remove the expressiveness limitations of this prior result, extending the search space construction, preconditioning and traversal algorithms to arbitrary multidimensional affine schedules.

7. Conclusion

Present day compilers fail to model the complex interplay between different optimizations and their effect on code on all the different processor architecture components. Empirical search has become essential to achieve portable high performance in spite of the analytically intractable hardware complexity. Most iterative compilation techniques target compiler optimization flags, parameters, decision heuristics, or phase ordering [10, 14, 41, 24, 42, 1]. We take a more aggressive stand, aiming for the construction and tuning of complex sequences of transformations.

Affine schedules build a very expressive search space, since a single schedule can represent an arbitrarily complex sequence of loop transformations. The first attempts to traverse such a space faced legality problems and showed poor results because only few legal affine schedules were found [31, 29]. Pouchet et al. recently proposed a solution for a restricted class of loop nests and transformations [35]. This paper targets *all* static control programs and, by construction, enables iterative optimization in a *closed space of semantics-preserving transformations*. To overcome the combinatorial nature of the optimization search space, we designed heuristics and a genetic algorithm with specialized operators that leverage the algebraic properties of this space, embedding the legality constraints into the operators themselves. We simultaneously demonstrate good performance gains and excellent convergence speed on huge search spaces, even on larger loop nests where fully iterative affine scheduling has never been attempted before.

For future work, we intend on looking at automatic parallelization, and at incorporating loop tiling for both locality and parallelism. While the back-end compiler may support some form of tiling, early experiments show that applying it as a post-pass does not significantly improve performance. Complexity of the generated code is partly responsible for this.

Acknowledgments Uday+Ram+Saday for fruitful talks and Cornero/Rohou for the traviata board

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 *Proc. of the Intl. Symposium on Code Generation and Optimization (CGO'06)*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *SC'2000 High Performance Networking and Computing*, Dallas, november 2000.

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

[4] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[5] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic, 1993.

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

[7] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, september 2004.

[8] 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.

[9] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.

[10] 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 *W. on Profile and Feedback Directed Compilation*, Paris, October 1998.

[11] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Affine transformations for communication minimal parallelization and locality optimization of arbitrarily nested loop sequences. Technical Report OSU-CISRC-5/07-TR43, Ohio State Univ., 2007.

[12] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–77, New York, NY, USA, 2005. ACM Press.

[13] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9,

Atlanta, Georgia, July 1999. ACM Press.

[14] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomputing*, 23(1):7–22, 2002.

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

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

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *Intl. Journal of Parallel Programming*, 21(5):313–348, october 1992.

[18] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. Journal of Parallel Programming*, 21(6):389–420, december 1992.

[19] 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.

[20] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[21] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[22] W. Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, Univ. of Maryland, 1996.

[23] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[24] 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.

[25] M. Le Fur. Scanning parameterized polyhedron using fourier-motzkin elimination. *Concurrency - Practice and Experience*, 8(6):445–460, 1996.

[26] C. Lee. UTDSP benchmark suite, 1998. `http://www.eecg.toronto.edu/~corinna/DSP`.

[27] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *ACM Symp. on Principles of Programming Languages (PoPL'97)*, pages 201–214, New York, NY, USA, 1997. ACM Press.

[28] S. Long and G. Fursin. Systematic search within an optimisation space based on unified transformation framework. *IJCSE Intl. Journal of Computational Science and Engineering*, 2006.

[29] S. Long and M. O'Boyle. Adaptive Java optimisation using instance-based learning. In *Proc. ICS*, pages 237–246, 2004.

[30] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *ACM Symp. on Principles of Programming Languages (PoPL'93)*, pages 2–15, Charleston, South Carolina, Jan. 1993.

[31] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *HPCN Europe 1998: Proceedings of the Intl. Conf. and Exhibition on High-Performance Computing and Networking*, pages 987–989, London, UK, 1998. Springer-Verlag.

[32] M. Palkovič. *Enhanced Applicability of Loop Transformations*. PhD thesis, T. U. Eindhoven, The Netherlands, Sept. 2007.

[33] 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.

[34] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. A note on the performance distribution of affine schedules. 2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'08), Göteborg, Sweden, January 2008.

[35] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Intl. Conf. on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, Mar. 2007.

[36] 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.

[37] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

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

[39] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. *SIGPLAN Notices, Proceedings of the 2007 PLDI conference*, 42(6):405–414, 2007.

[40] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[41] 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.

[42] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.

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

[44] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, september 2007. To appear.

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

[46] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

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

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

[49] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.