A Note on the Performance Distribution of Affine Schedules

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

[1] ALCHEMY Group, INRIA Futurs and University of Paris-Sud 11,
first.last@inria.fr
[2] Computer and Information Sciences, University of Delaware,
cavazos@cis.udel.edu

Abstract. Iterative optimization has been shown to improve the performance of benchmarks significantly, but its application involves challenges such as the requirement of an expressive search space and the design of efficient search techniques. In this paper, we apply iterative optimization to the problem of optimizing in the polyhedral model, a powerful algebraic representation of any static control program, by using affine multidimensional schedules to represent arbitrarily complex transformation sequences. We propose to study the performance distribution of the search space of affine multidimensional schedules built specifically to guarantee legality and uniqueness of each program version. We extensively study the optimization of 5 representative benchmarks in this representation, and highlight a series of static and dynamic characteristics of the search space. We show how the space can be decoupled into subspaces, which can be statically ordered with respect to their impact on performance. Finally, we present a practical search method leveraging these properties to traverse the search space, yielding a 32.56% speedup on eight representative kernels.

Keywords Iterative optimization, polyhedral model, affine scheduling, loop transformations.

1 Introduction

Loop nest optimizations are some of the most important compiler transformations to fully exploit all the features of a given architecture. Unfortunately, as architecture complexity grows, static optimization techniques hard-wired in modern compilers usually fail to achieve the maximum performance for a given program. In recent years, iterative optimization has proven to be successful at addressing this issue. The main idea of iterative optimization is to optimize and run several versions of a program on the target architecture, searching for the best set of transformations for each specific program. This search is directed by the actual performance feedback of all tested versions, in contrast to an analytical model which typically is not accurate enough to reflect both architecture and compiler complexity.

Iterative optimization has two inherent challenges: (a) the construction of an expressive search space that includes the best performing program versions; and

(b) the design of efficient search mechanisms to traverse this space. Approaches proposed to date have faced problems with illegal program versions being generated [8] and only covering some compiler options or a fraction of the available set of possible loop transformations [1, 7]. We propose to tackle these problems by considering iterative optimization in the context of the *polyhedral model* [11, 4], a powerful algebraic representation of *any static control program*. Within this model, one can represent an arbitrarily complex sequence of loop transformations *with a single transformation*, namely a transformation in an affine schedule representation of the program [4–6]. Moreover, critical properties such as legality or uniqueness of the generated version can be directly modeled in the search space, dramatically improving the convergence of search techniques [10, 9].

In this paper, we propose to study and characterize the performance distribution of a search space of multidimensional schedules by means of statistical analysis of results over an extensive set of program versions. We show how the search space can be decoupled into subspaces ordered with respect to their impact on performance. We build upon these results to motivate an heuristic mechanism to traverse such a search space, leveraging various static properties of the space. Finally, we show that this technique enables us to discover significant speedups in a very limited number of runs.

The paper is organized as follows. Section 2 describes how to construct a search space encompassing only legal, distinct program versions. In Section 3, we study the performance distribution of 5 representative UTDSP kernels. Next, we derive from this study an efficient heuristic search mechanism in Section 4. Finally, we conclude in Section 5.

2 Generating Program Versions

Programs are represented in the polyhedral model by means of three components for each statement in the original code. First, an *iteration domain* exactly describes the set of executed instances of each statement (typically statements enclosed in loops). Second, a set of array *subscript functions* describes, over the iteration domain, each array index accessed by the statement. Third, a multidimensional schedule (which assigns a multidimensional timestamp to each instance of the statement) specifies the execution order of all those instances. This schedule is represented by a matrix Θ whose coefficients are called the *schedule coefficients*. Each of these three components are affine forms: more complex forms are beyond the scope of our tool. Finally, source code for the program can be regenerated from the polyhedral representation through the use of efficient code generation algorithms [2]. The reader can refer to [4, 5, 10] for a comprehensive description of this representation.

While *any loop transformation* can be represented by means of affine multidimensional schedules [11], we choose to limit this study to the more tractable compositions of interchange, skewing, reversal, fusion, distribution, peeling and shifting (and indirectly index-set splitting).[3]

[3] This does not forbid the compiler to apply any other transformation, we simply do not model them in our representation.

2.1 Building the Search Space

It is not possible to apply any scheduling function to a program without changing its semantics. Scheduling must preserve the relative order of dependent statement instances to preserve the original program semantics. Choosing a schedule at random is likely to lead to an illegal program version, i.e., one that changes the semantics of the program, since the probability of finding a *legal* schedule (which does not alter semantics) decreases exponentially fast with the length of the program [10]. Previous works on using a polyhedral approach for iterative compilation showed poor results, in particular, because they did not consider the data dependence problem early enough [7, 8].

Basically, two statement instances are in a dependence relation if they access the same memory location and at least one of these accesses is a write [3]. To ensure that a schedule does not alter the semantics, it has to satisfy the *precedence constraint*, for all pairs of instances which are in a dependence relation:

$$\theta_R(\boldsymbol{x}_R) \prec \theta_S(\boldsymbol{x}_S)$$

Where $\prec$ denotes the *lexicographic ordering*,[4] θ_R (θ_S) is the multidimensional affine schedule of a statement R (S), $\boldsymbol{x}_R$ ($\boldsymbol{x}_S$) is an iteration vector of the statement (that is, $\boldsymbol{x}_R$ ($\boldsymbol{x}_S$) takes any value in the iteration domain), and there is a dependence $S \rightarrow R$. Dependences in a *static control part* (SCoP) are exactly expressed by *dependence polyhedra* whose formal description has been proposed by Feautrier [4]. A SCoP is a maximal set of consecutive statements where the control flow and data accesses can be fully described at compile time (by affine forms, in the polyhedral model).

In multidimensional schedules, some dependences may be entirely satisfied[5] at a given dimension d of the schedule (the d^{th} row of θ). We refer to these dependences as *strongly solved* at dimension d. In contrast, other dependences may still have some points such that their schedule are only equal for the d first dimensions, we call them *weakly solved* dependences at dimension d [5].

Building a search space of affine multidimensional schedules reveals to two combinatorial problems. First, there exist many distinct solutions to the problem of *deciding* which dependences will be strongly or weakly solved at a given dimension, and each of them leading to potentially different search spaces. Second, the constructed search spaces are too large to be explored exhaustively for complex programs.

Feautrier found a very interesting solution with the space of all legal schedules leading to maximum fine-grain parallelism [5]. To achieve this, he proposed a greedy algorithm to maximize the number of dependences strongly solved for a given schedule dimension. This solution is interesting because it tends to reduce the number of schedule dimensions — hence reducing the search space size — and exhibits parallelism that one may exploit. But this greedy algorithm faces a lack

[4] $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ 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 $a_i < b_i$.

[5] For dependences to be satisified, all instances in the dependence relation must respect the precedence constraint.

of scalability due to the size of the integer programs to be resolved. Moreover, maximal reduction of the schedule dimension is likely to translate into large scheduling coefficients which are known to be a source of heavy control overhead when generating the target code for sequential targets [2]. Finally, we want to explore several possible values for a schedule, and not limit ourselves to a single solution computed with the help of a cost function.

We suggest a simple variation to overcome these 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 by performing instancewise analysis [4]. Mark all dependences as weakly solved. Initialize dimension to $d = 1$.
2. While all dependences are not strongly solved:
   (a) Initialise $\mathcal{L}_d$, the space of legal schedules for dimension d, to universe ("full" polyhedron).
   (b) For each non-solved dependence $\mathcal{D} \in G$:
      – Express the space $\mathcal{T}_\mathcal{D}$ of legal schedules respecting the precedence constraint for $\mathcal{D}$ and not violating any other dependences in G (see Pouchet et al. for a formal description of $\mathcal{T}_\mathcal{D}$ computation [10]).
      – If $\mathcal{L}_d \cap \mathcal{T}_\mathcal{D}$ is not empty, then $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{T}_\mathcal{D}$ and mark $\mathcal{D}$ as strongly solved.
   (c) Remove all strongly solved dependences from G, $d \leftarrow d + 1$ and go to 2.

This algorithm outputs for each schedule dimension d a space $\mathcal{L}_d$ of legal values for the d^{th} row of Θ. Intuitively, a schedule of dimension s means that the s outer most loops of any loop nest in the generated program will be sequential loops (this implies there exists $depth - s$ level of parallelism in the generated program, where $depth$ is the maximal loop depth in the SCoP).

The algorithm terminates since at least one dependence can be strongly solved per dimension [5]. It differs from the algorithm proposed by Feautrier as it does not guarantee to maximize the number of dependences solved per dimension. It follows that it may not minimize the schedule dimensionality. However, this algorithm is efficient and only needs one polyhedron emptiness test[6] for each dependence. Since in this study we address the problem of optimizing sequential codes, exposing parallelism is less critical and *we bound the coefficient values between* $[-1, 1]$ *to avoid control overhead at code generation* [2]. While this may restrict us from finding any solution if we are constrained to one-dimensional schedules [10], it may just translate to additional dimensions on multidimensional schemes. Hence, this solution gives an interesting trade-off between scalability, efficiency, and parallelism extraction for further studies.

2.2 Picking Points in the Space

The algorithm presented in Section 2.1 constructs a polytope $\mathcal{L}_d$ per sequential schedule dimension d, for a given program. Our iterative optimization must traverse these polytopes to build different legal versions of the original program, hence, we must provide efficient mechanisms for this traversal.

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

Each program version is represented by a unique scheduling matrix Θ. First, are schedule coefficients attached to each loop iterator surrounding a statement in the original program, for all statements ($\boldsymbol{i}$). Second, are schedule coefficients attached to global parameters ($\boldsymbol{p}$), for all statements. Third, are the schedule coefficients attached to 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 $\boldsymbol{x}$:

$$\Theta.\boldsymbol{x} = \begin{pmatrix} \boldsymbol{i}_1^1 \cdots \boldsymbol{i}_l^1 \, \boldsymbol{p}_1^1 \cdots \boldsymbol{p}_l^1 \, c_1^1 \cdots c_l^1 \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ \boldsymbol{i}_1^s \cdots \boldsymbol{i}_l^s \, \boldsymbol{p}_1^s \cdots \boldsymbol{p}_l^s \, c_1^s \cdots c_l^s \end{pmatrix} . (\, \boldsymbol{x}_1 \cdots \boldsymbol{x}_l \, \boldsymbol{n}_1 \cdots \boldsymbol{n}_l \, 1 \cdots 1 \,)^T$$

To build each row Θ_d, we apply a dynamic scanning over the legal polytope $\mathcal{L}_d$, by *sequentially* picking values for each coefficient in a fixed order. We first shape the polytopes by computing their complete projection,[7] thanks to a modified Fourier-Motzkin algorithm improved for scalability [9].

It is possible to efficiently complete or even correct any vector, i.e. slightly modify its coordinates to make it lie in the polytope of legal schedules. This simple correction procedure works as follows. Given a vector $\boldsymbol{v}$ of size n, for $i \in [1, n]$, first, compute the lower bound and upper bound of v_i, provided the values for $v_1 \cdots v_{i-1}$ and second, if $v_i \notin [lb, ub]$, then $v_i = lb$ if $v_i < lb$ or $v_i = ub$ if $v_i > ub$. Hence it is possible to partially build a schedule prefix (e.g., pick values for the $\boldsymbol{i}$ coefficients and set all other coefficients to 0) and applying this correction principle on it will result in finding the minimal amount of complementary transformations needed to make this transformation sequence legal. We refer to this mechanism as the *completion algorithm*.

Three fundamental properties are embedded in this completion algorithm. Given a point $\boldsymbol{v}$:

1. provided a legal value for $v_1 \cdots v_i$, a completion always exists;
2. this completion will only update $v_{i+1} \cdots v_n$, if needed;
3. the smallest legal value for the $\boldsymbol{p}$ coefficients (i.e. the minimal correction of the 0 value) translates into the maximal fusion available *for the picked* $\boldsymbol{i}$ *coefficients*.

In practice this algorithm is extremely powerful: it allows us to build only parts of the schedule, focusing on some of its properties, and the heuristic will automatically complete the schedule with a minimal amount of correction to make it legal. It also motivates the order of coefficients in the Θ matrix. We assert that the most performance impacting transformations (interchange, skewing, reversal) are embedded in the first coefficients of Θ — the $\boldsymbol{i}$ coefficients; followed

[7] It implies the order to compute coefficient values is the reverse order of the variable projection, that is from $\boldsymbol{i}_1$ to c_p

by coefficients involved in fusion and distribution — the $\boldsymbol{p}$ coefficients; and finally the less impacting c coefficients, representing loop shifting and peeling. The correction algorithm will find complementary transformations in order of least to most impacting, as it will not correct any prefix if a legal suffix exists.

3 Performance Distribution

The polyhedral representation of programs offers a compact way to represent arbitrarily complex sequences of transformations, significantly increasing the expressiveness of the search space. Moreover, the design of traversal methods for such spaces is facilitated by the algebraic properties of the model. For instance it is possible to consider only legal sequences, dramatically narrowing the search. We propose to go deeper and expose static characteristics of the space correlated to performance distribution. We extensively study the performance distribution of some representative benchmarks to assess the following hypotheses.

1. It is possible to statically order the impact on performance of transformation coefficients, that is, decompose the search space in subspaces where the performance variation is maximal or reduced.
2. The more a schedule dimension impacts a performance distribution, the more it is constrained.

As a result of this hypothesis, traversal techniques can be designed to focus on the most promising subspaces first, notably increasing the efficiency of the search method.

3.1 Experimental Protocol

For each tested point of the search space, we generate the corresponding C code with `ClooG` [2], add all the required instrumentation to the code, then compile and run it on the target machine. Our target architecture is an AMD Athlon X64 3700+ (single core), running at 2.4GHz (configured with 64KB+64KB L1 cache and 1024k L2 cache). The system is Mandriva Linux and the native compiler is GCC 4.1.2. All generated programs (as well as the original codes) were compiled using the following optimization settings, known to bring excellent performance for this platform: `-O3 -msse2 -ftree-vectorize`. The performance data are collected using hardware counters, using the PAPI library. We collected counters for cycles, L1 and L2 hits and misses, and branches taken and mispredicted. To limit OS interference to the minimum, all program versions are run with real-time priority scheduler and averaged over 100 executions.

3.2 Study of the `dct` Benchmark

The `dct` benchmark presented in Figure 1 computes a 32x32 Discrete Cosine Transform (M=32). This well known kernel is a good candidate for aggressive optimizations, and representative of several challenges for compilers. It is imperfectly nested, has 35 dependences and exposes possible multi-level fusion. Also, the `cos1` array can be reused, by means of a complex transformation sequence.

The `latnrm` benchmark presented in Figure 2 is a normalized lattice filter, and will be studied in Section 3.3.

```
for (i = 0; i < M; i++) {
  for (j = 0; j < M; j++) {
    temp2d[i][j] = 0.0;
    for (k = 0; k < M; k++) {
      temp2d[i][j] += block[i][k] *
                      cos1[j][k];
    }
  }
}
for (i = 0; i < M; i++) {
  for (j = 0; j < M; j++) {
    sum = 0.0;
    for (k = 0; k < M; k++) {
      sum += cos1[i][k] * temp2d[k][j];
    }
    block[i][j] = ROUND(sum2);
  }
}
```

Fig. 1. Source Code for `dct`

```
for (i = 0; i < M; i++) {
    top = data[i];
    for (j = 1; j < N; j++) {
        left = top;
        right = internal_state[j];
        internal_state[j] = bottom;
        top = coefficient[j-1] * left -
              coefficient[j] * right;
        bottom = coefficient[j-1] * right +
                 coefficient[j] * left;
     }
    internal_state[N] = bottom;
    internal_state[N+1] = top;
    sum = 0.0;
    for (j = 0; j < N; j++)
      sum += internal_state[j] *
             coefficient[j+N];
    outa[i] = sum;
  }
```

Fig. 2. Source Code for `latnrm`

Statistics of the search space for `dct` The space of legal affine multidimensional schedules is built according to the algorithm presented in Section 2.1. This technique builds a search space where 3 sequential dimensions are necessary to respect the program dependences (the Θ matrix has 3 rows). Its statistics are summarized in Figure 3. For each schedule dimension, we report the *degree of freedom* (that is, the number of *different legal schedules*) decomposed in 3 different classes. The $\boldsymbol{i}$ class represents all the schedules with a distinct $\boldsymbol{i}$ prefix (that is, where iterator coefficients are going to be different, typically distinct legal combinations of interchange, skewing, reversal); then respectively for the $\boldsymbol{i}+\boldsymbol{p}$ class (adding fusion, distribution); and the $\boldsymbol{i}+\boldsymbol{p}+c$ class (adding peeling, shifting). Finally, the size of the search space for the entire program is shown in the Total combined row, for each 3 classes (multiplying the degree of freedom for each schedule dimension).

Schedule dimension	$\boldsymbol{i}$	$\boldsymbol{i}+\boldsymbol{p}$	$\boldsymbol{i}+\boldsymbol{p}+c$
Dimension 1	39	66	471
Dimension 2	729	19683	531441
Dimension 3	60750	1006020	64855485
Total combined	1.7×10^9	1.3×10^{12}	1.6×10^{16}

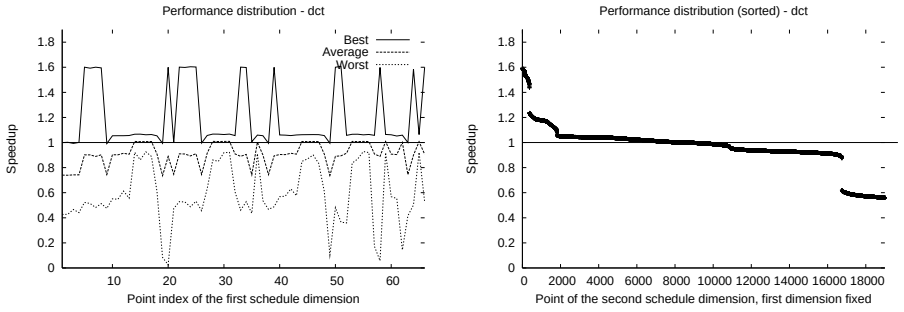
Fig. 3. Search Space Statistics for `dct`

It is worth recalling that each program version corresponds to an arbitrarily complex sequence of transformations applied to the original program. It is possible to limit the degree of freedom to (a part of) the $\boldsymbol{i}$ or $\boldsymbol{i}+\boldsymbol{p}$ classes, by simply relying on our completion algorithm to find the minimal set of complementary

transformations (contained in the larger classes) to make the current sequence legal. In this case we do not explore the numerous possibilities to make this sequence legal, but instead use the completion algorithm to generate only one of them.

Performance distribution To limit the set of tested program versions, we rely upon two empirical observations. First, it is expected that the degree of freedom for peeling and shifting will have a low impact on the performance distribution [10], hence we can safely limit the traversal to the $\boldsymbol{i} + \boldsymbol{p}$ class. Second, for the `dct` benchmark, it is expected that the third schedule dimension will have a low impact on performance: it will only affect the inner-most scheduling of two statements with a regular memory pattern, thus very little improvement can be expected.[8] Eventually, we consider a search space of 1.29×10^6 different program versions, where each schedule coefficient that is not explored is computed with the completion algorithm.

Figure 4 shows the performance distribution of all versions generated for the `dct` program. Figure 4(a) plots the *best*, *worst*, and *average* performance for each of the 66 possible values for Θ_1 (represented in the x axis). For each of these values, we evaluated the 19683 possible values for Θ_2, and reported the performance. The performance of the original code is represented by the bold horizontal bar: each point above this bar improves the original code. Figure 4(b) plots the raw performance (sorted from the best to the worse) of all the 19683 points of Θ_2, using the value of Θ_1 of the best found version.



(a) Representatives for each point of Θ_1

(b) Raw performance of each point of Θ_2, for the best value for Θ_1

Fig. 4. Performance Distribution for `dct`

The first observation is that an important speedup can be discovered: the best optimization achieves a speedup of 61.7%. Also, as what was pointed out in [10], several program versions achieve a similar performance.

[8] We performed sampling also in the $\boldsymbol{i} + \boldsymbol{p} + c$ class as well as for the third schedule dimension: it always confirmed these assumptions.

The difficulty to reach the best improving points in the search space is emphasized by their extremely low proportion: only 0.14% of points achieves at least 80% of the maximal speedup, while only 0.02% achieves 95% and more. Conversely, 61.11% degrades performance of the original code, while in total 10.88% degrades the performance by a factor 2. Hence in this context it is expected that pure random approaches will fail to converge quickly to the best speedup.

We note that there are several values for the first schedule dimension from which it is impossible to attain the maximal performance. However, the maximal performance is attainable from more than one point in the first dimension. We conclude that effectively searching for points in Θ_1 is important in obtaining good performance, but cannot be the only criterion in designing search techniques when performing iterative optimization in the polyhedral representation.

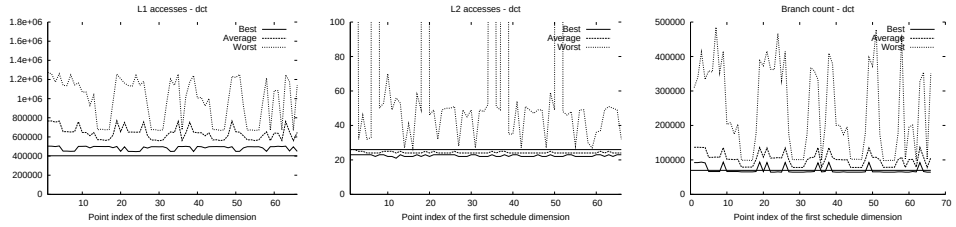
Statistical analysis This section describes a finer grain analysis by capturing the relative impact of the schedules coefficients on the performance distribution. We first compute the variance of each schedule coefficient on the set of versions achieving at least 80% of the maximum speedup. Coefficients with little to no variance among points with good speedups means that those coefficients are important in obtaining that good performance. We observe that 7 (out of 12) coefficients of the $\boldsymbol{i}$ class of Θ_1 have the same value, as well as 2 (out of 5) coefficients of the $\boldsymbol{p}$ class. Also, 3 (out of 12) coefficients of the $\boldsymbol{i}$ class of Θ_2 have a very low variance, emphasizing that the second dimension Θ_2 plays an important role in obtaining a good characterization of the performance distribution. The impact of the coefficients with low variance on the complete distribution shape is confirmed by correlating the performance of a program version with a non-optimal value of these coefficients. For example, we found that slight changes to any of the $\boldsymbol{i}$ low variance coefficients of Θ_1 translated into major performance variations.

We observe that a relevant ordering of the impact of the several classes of coefficients is $\boldsymbol{i} < \boldsymbol{p} < \boldsymbol{c}$, and studying the variance of the coeffecients confirmed our first hypothesis stated in Section 3.

Hardware counters details Figure 5 gives more details on source of performance improvements and degradations. It reports the behavior for the *L1 accesses*, *L2 accesses* and *Branch count* metrics.[9] The performance of the original code is represented by a bold horizontal bar.

The metric that seems to capture the performance distribution shape best is the *L1 accesses* curve. We observe that all transformations access the L1 cache more than the original code does. The transformed code performs at least 8% more L1 accesses than the original code. Also, the lowest L1 accesses points corresponds exactly to the peaks of highest speedup reported in Figure 4(a). On the other hand, several transformed program versions access the L2 cache less than the original code. Hence, the criterion in terms of memory accesses for optimal performance is to minimize L1 and L2 accesses. Note, we increase the number

[9] Accesses = hits + misses, count = taken + mispredicted



(a) L1 Accesses (b) L2 Accesses (c) Branch Count

Fig. 5. Hardware Counters Distribution for `dct`

of L1 accesses as compared to the original code because there are more hits to the L1 cache, thereby minimizing L2 accesses. The last reported performance counter statistic is Branch count. We can correlate this statistic with the control statements added in the transformed code. The polyhedral code generation algorithms are likely to generated many complicated control statements (if and modulos) when highly complex transformations are applied. While not directly correlated to the performance distribution itself, this metric shows that the space contains many complicated versions, and in most cases a transformation sequence leads to more branches than the original code.

Discussion of the performance counter statistics The best performing transformations reduce 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 not obvious 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. Overall, our results confirm the potential of iterative optimization to find program versions that better exploit the complex behavior of current superscalar processors. Also, we have extended iterative optimization to optimization problems far more complex than those commonly solved in adaptive compilation.

3.3 Evaluation of Highly Constrained Benchmarks

We established in the previous sections a connection between the $\boldsymbol{i}$ class and the dispersion of the performance distribution, on a representative benchmark offering a large degree of freedom for scheduling. In this section, we study the influence of a strong limitation of the degree of freedom of the $\boldsymbol{i}$ class. In particular, such a situation may derive from the greedy algorithm of Section 2.1 which tends to reduce the degree of freedom for the $\boldsymbol{i}$ class of the first dimension.

Search space statistics on more examples In the following we focus on four representative benchmarks extracted from the UTDSP suite. Namely `latnrm`, a

normalized lattice filter (shown in Figure 2); `fir`, Finite Impulse Response filter; `lmsfir`, a Least Mean Square adaptive FIR filter; and `iir`, an Infinite Impulse Response filter. Figure 6 shows the search space statistics for the first schedule dimension for the $\boldsymbol{i}$, $\boldsymbol{p}$, and c classes. We also report, for each benchmark, the number of statements (# St.), the number of dependences (# Deps.) and the number of schedule dimensions (# Dim.) needed to represent the program.

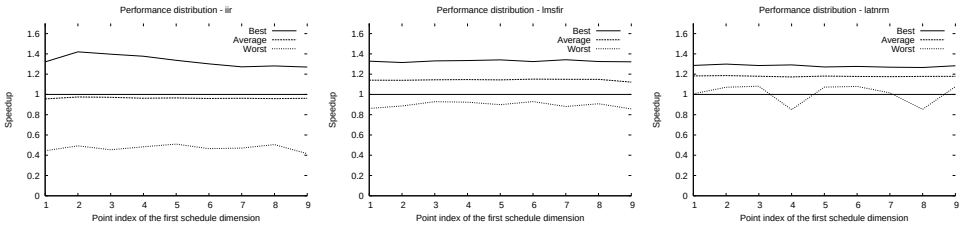
Benchmark	# St.	# Deps.	# Dim.	$\boldsymbol{i}$	$\boldsymbol{i}+\boldsymbol{p}$	$\boldsymbol{i}+\boldsymbol{p}+c$
`latnrm`	11	75	3	1	9	27
`fir`	4	36	2	1	9	18
`lmsfir`	9	112	2	1	9	27
`iir`	8	66	3	1	9	18

Fig. 6. Search Space Statistics

We observe for each benchmark that the degree of freedom of the $\boldsymbol{i}$ class, for the first dimension, is null: there is only one sequence of interchange, reversal and skewing available for the first schedule dimension in the search space. This situation is not connected to usual program indicators such as number of statements or dependences, it is necessary to build the search space to detect this static property. Moreover, the four benchmarks we consider are syntactically different, and representative of many kernels in embedded computing.

We show in the following how this lack of degree of freedom translates into regularities of the performance distribution, and in performance improvements.

Performance distribution We conducted for each of these benchmarks the same study as presented in Section 3.2. We exhaustively traverse the $\boldsymbol{i}+\boldsymbol{p}$ class, for the first two schedule dimensions. Figure 7 shows the performance distributions from this search.



(a) `iir` (b) `lmsfir` (c) `latnrm`

Fig. 7. Performance Distribution for 3 UTDSP benchmarks

Again, we see there is significant speedup to be discovered: more than 30% speedup can be achieved for each of these benchmarks. Hence, for these bench-

marks, the limited degree of freedom of the first schedule dimension does not restrict significant speedups.

The performance distribution is almost flat, another evidence of the impact of transformation coefficient freedom. We can conclude that the degree of freedom in the $\boldsymbol{i}$ class translates into variations in the performance distribution.

We also conducted variance studies to capture the relative impact of schedule coefficients. We observe that the impact on performance distribution of the $\boldsymbol{p}$ coefficients is lower than the $\boldsymbol{i}$ ones, while the impact of the c coefficients is almost negligible. Overall, all the conducted experiments confirm our initial hypothesis from Section 3.

3.4 Addressing the Generalization Issue

In order to assess the hypothesis that schedule coefficients can be ordered with respect to their impact on performance, we need to distinguish two different concerns: the generalization to other programs, and the generalization to other architectures.

Generalization to other programs We showed a positive correlation between the amount of variation in the performance distribution and the degree of freedom in the $\boldsymbol{i}$ class, especially for the first schedule dimension. Hence, it is expected that a traversal of each possible value for this class will be necessary to guarantee to achieve the maximal performance. Nevertheless, in the general case, particular fusions and distributions can achieve a dramatic impact on performance, and the full $\boldsymbol{i}+\boldsymbol{p}$ class is of interest for traversal. Finally, traversing the degree of freedom for peeling and shifting is almost useless, as the aim of those transformations usually is program legality and not program performance.

Generalization to other architectures Generalizing the results obtained on AMD Athlon64 to other architectures must be done with care. First, even if it is clear that the $\boldsymbol{i}$ class will still have a large impact on performance regardless of the architecture, one has to pay attention to fusion and distribution: which are important transformations for several embedded architectures. Hence, motivating the traversal of the degree of freedom offered by the $\boldsymbol{i}+\boldsymbol{p}$ class. We also conducted experiments on the ST231 embedded VLIW processor, though different from the AMD Athlon, we still observed a similar impact of the $\boldsymbol{i}$ class to the shape of the peformance distribution. Although smaller speedups were found (the regular VLIW architecture is easier to model and well exploited by the STMicroelectronics compiler), our framework is still able to discover performance improvements on all tested benchmarks, with an average of 13.3%.

Performance distribution discussion From this study of the performance distribution of several programs, we deduce the following facts.

1. The degree of freedom in the $\boldsymbol{i}$ class of Θ_1, the first row of Θ, translates into variation in the performance distribution.
2. When the degree of freedom in the $\boldsymbol{i}$ class of Θ_1 is nonexistent, the performance distribution is almost flat.

3. The impact of coefficients on performance is ordered: Θ_1 impacts performance more than Θ_2, and inside a schedule row, $\boldsymbol{i}$ coefficients impacts performance more than $\boldsymbol{p}$ and c.

We leverage these characteristics of the search space to motivate the design the traversal heuristic introduced in our previous work [9]. We detail the traversal heuristic, and present its performance on 8 benchmarks.

4 Efficient Search Space Traversal Heuristic

The traversal approach we propose relies upon the computation of the degree of freedom available for the $\boldsymbol{i}$ class of Θ_1 (refered to as $card(\mathcal{L}_1^{\boldsymbol{i}})$). Depending on its value, the heuristic will apply more or less of the available transformation sequences. In addition, we apply a feedback-driven filter. Experiments showed that Θ_1 is a good indicator of the overall performance distribution, hence we can start by traversing several values for Θ_1 with a unique value for the remaining Θ_k, and in a second step traverse several values for Θ_k provided the best found value(s) for $\Theta_{(1\ldots k-1)}$. Finally, we rely on our completion algorithm to compute the schedule coefficients which are not explored.

The traversal heuristic can be depicted as follows:

1. $\Theta_k = completion(\mathbf{0}, \mathcal{L}_k)$, for all row k of Θ
2. $d = 1$
3. if $card(\mathcal{L}_d^{\boldsymbol{i}+\boldsymbol{p}}) < L_1$, $\mathcal{L}_d^{'} = \mathcal{L}_d^{\boldsymbol{i}+\boldsymbol{p}}$ else $\mathcal{L}_d^{'} = \mathcal{L}_d^{\boldsymbol{i}}$
4. For each point $\boldsymbol{v}$ in $\mathcal{L}_d^{'}$
   (a) $\Theta_d = completion(\boldsymbol{v}, \mathcal{L}_d)$, keep $\Theta_k, k \neq d$
   (b) Evaluate Θ (generate code, compile and run)
5. Keep Θ_d from the best performing schedule Θ, $d = d + 1$, go to 3

The parameter L_1 drives the size of the search space explored: the larger the degree of freedom, the slower the convergence. By limiting to the $\boldsymbol{i}$ class typically for inner schedule dimensions, we target only the most promising subspaces at the expense of possibly missing the optimal program version. Also, one may note that this heuristic approach can be coupled with a static limit. In our experiments, we used a value of 50 for L_1 and a static limit of 1000 program versions evaluated.

Figure 8 shows the results for `dct` along with seven other kernels from the UTDSP suite that were amenable to the polyhedral representation without code modification.[10] We report the number of statements (# Stm.), the size of the $\boldsymbol{i}$ class for the first schedule dimension, the size of the search space considered (Space), the run number at which the best performing version was found (Id Best: the lower, the earlier), and the speedup achieved (Speedup). The overhead of picking a point in the search space and building its syntactic representation is negligible in comparison to the execution time of the program version, and several points can be tested within a second. Finally, the procedure is fully automated.

[10] `matmult` is a 2 statement matrix multiplication, for 10×10 matrices. See [10] for an extensive study of this kernel

	dct	matmult	lpc	edge-c2d	iir	fir	lmsfir	latnrm
#Inst.	5	2	12	3	8	4	9	11
i	39	76	243	1	1	1	1	1
Space	1.6×10^{16}	912	$> 10^{25}$	5.6×10^{15}	$> 10^{19}$	9.5×10^{7}	2.8×10^{8}	$> 10^{22}$
Id Best	46	16	489	11	34	33	51	6
Speedup	57.1%	42.87%	31.15%	5.58%	37.50%	40.24%	30.98%	15.11%

Fig. 8. Heuristic Performance for AMD Athlon

The heuristic succeeds in discovering an average speedup of 32.56% for the 8 tested benchmarks. All the best versions are the result of the application of a complex transformation sequence, syntactically very different from the original code. Analysis of the performance counters for these transformations shows improvements in memory behavior, combined with a better workload of the processor units which is likely to be the result of hard to predict interaction between the compiler optimizations and the processor features.

The limited performance improvement for `edge` 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.

For the case of highly constrained benchmarks, we also specifically studied the performance of a single statically computed schedule: the one computed at step 1 of the traversal heuristic (i.e. the application of the completion algorithm on all schedule coefficients). This schedule performs extremely well, and succeeds in discovering 75%–99% of the maximum speedup available in the space, *without evaluation*. Hence, the proposed heuristic can be coupled with the detection of the special case where $card(\mathcal{L}_1^{\boldsymbol{i}}) = 1$ to avoid traversing a space leaving few room for further improvements (as it is expected that the performance distribution will be almost flat). This approach leads to a an average 17.8% speedup on the 5 benchmarks where this criteria applies, without any further evaluation required.

5 Conclusion

Analytical approaches to drive compiler optimizations fail to model the complex interplay between hardware and compiler features. Iterative optimization techniques has become essential in overcoming the problems with model. However, high-level loop transformations challenge the design of such methods because 1) *expressing* complex restructuring sequences is a hard problem and 2) efficient methods to *traverse* huge and complex search spaces have to be designed.

Applying iterative optimization to the polyhedral model provides a significant breakthrough to the challenges of expressiveness and applicability. It enables searching in a space where every point is relevant: each point corresponds to a legal, distinct program version resulting in the application of an arbitrarily complex sequence of transformations [9, 10].

To explore the construction of efficient space traversing techniques, we present a detailed analysis of the performance distribution of affine multidimensional schedules. We build upon this analysis to characterize performance distributions relating static and dynamic properties of different program versions. From the static point of view, we show that a relative ordering of the subspaces (i.e., the different classes of coefficients of the schedule) of the search space can be achieved with respect to their impact on performance. We show that it dramatically helps to narrow the search to the subspaces that impact performance the most. ¿From the dynamic point of view, we show that the number of program versions that achieve a significant fragment of the maximal speedup available is extremely low. We build upon these static and dynamic characteristics to construct a powerful heuristic traversal method that exposes an average speedup of 32.56% on 8 representative kernels on an AMD Athlon64.

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, 2006.
2. 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.
3. A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.
4. 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.
5. 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.
6. W. Kelly. *Optimization within a Unified Transformation Framework.* PhD thesis, Univ. of Maryland, 1996.
7. S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *ICPPW '05: Proceedings of the 2005 Intl. Conf. on Parallel Processing Workshops (ICPPW'05)*, pages 137–144, Washington, DC, USA, 2005. IEEE Computer Society.
8. 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.
9. L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. Iterative optimization in the polyhedral model: Part II, multidimensional time. Nov. 2007. Submitted for publication, available on request.
10. 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.
11. W. Pugh. Uniform techniques for loop optimization. In *ICS'5 ACM Intl. Conf. on Supercomputing*, pages 341–352, Cologne, june 1991.