

A Note on the Performance Distribution of Affine Schedules

Louis-Noël Pouchet¹, Cédric Bastoul¹, John Cavazos², and Albert Cohen¹

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. Multidimensional affine schedules are a compact way to represent a variety of transformation sequences. Moreover, they allow an upstream characterization of some important properties such as legality or uniqueness of the target code, two fundamental properties known to be a bottleneck in iterative optimization.

We propose in this paper to study the performance distribution of a search space of affine multidimensional schedules built specifically to guarantee legality and uniqueness of each program versions. We extensively study the optimization of 5 representative benchmarks, and highlight a series of static and dynamic characteristics of the search space.

Finally we present a practical iterative method to efficiently traverse this search space for any static control program, yielding a 32.56% speedup on eight representative kernels.

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

1 Introduction

Program transformations and especially loop nest optimization are some of the most critical compiler techniques to take advantage of all features of a given architecture. Unfortunately, as the architecture complexity grows, static optimization techniques embedded in modern compilers usually fail to achieve the maximum performance for a given program. Iterative compilation is a matured framework addressing this issue. The basic idea is to iteratively generate and run several versions of a program on the target architecture. It is directed by the actual performance feedback of all tested versions, by opposition to any theoretical model which is typically not accurate enough to reflect both architecture and compiler complexity.

Iterative compilation challenges can be decoupled in (a) the construction of a search space including the most relevant program versions; and (b) the design of an efficient search mechanism to traverse this space. Approaches proposed so far face the bottleneck of program transformation applicability, e.g., legality,

and search space expressiveness: they only cover some compiler options or a fraction of the actual set of possible loop transformations [1, 7, 8]. We propose to tackle this problem 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 *within a single optimization*, namely an affine schedule for 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 observations over an extensive set of program versions. We build upon these results to motivate an heuristic mechanism to traverse such a space, leveraging on various static properties of this space. Finally, we show that this technique enables to discover significant speedups in a very limited number of runs.

The paper is organized as follows. First, we recall in Section 2 how to construct a search space encompassing only legal, distinct program versions. Then we extensively study the performance distribution of 5 representative UTDSP kernels in Section 3. 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. (1) An *iteration domain* exactly describes the set of executed instances of the statement. (2) A set of *subscript functions* describes (over the iteration domain) each array cell accessed by the statement. (3) A multi-dimensional schedule, which affects a multidimensional timestamp to each instance of the statement, specifies the execution order of all the statement instances. All these components are affine forms: more complex situations are beyond present day techniques. The reader may 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 case of arbitrary compositions of interchange, skewing, reversal, fusion, distribution, peeling and shifting (and indirectly index-set splitting).³

2.1 Building the Search Space

It is not possible to apply any scheduling function to a program without changing its semantics. Choosing a schedule at random will likely to lead to a non-valid target program as the probability of finding a *legal* (which does not alter semantics) schedule decreases exponentially fast as program’s length grows [10].

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

Previous works on using a polyhedral approach for iterative compilation showed poor results, in particular, because they didn't take the data dependence problem into consideration early enough [7, 8].

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

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

Where θ_R is the multidimensional affine schedule of a statement R , and \mathbf{x}_R is an iteration vector of the statement (that is, \mathbf{x}_R takes any value in the iteration domain). Dependences in static control parts (SCoPs) are exactly expressed by *dependence polyhedra* whose formal description has been proposed by Feautrier [4]. In multidimensional schedules, some dependences may be entirely satisfied (all instances in dependence relation respect the precedence constraint) at a given dimension d of the schedule (the d^{th} row of θ). They are called *strongly solved* dependences at dimension d . In contrast, some 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 .

We have to face two combinatorial problems. The first one is that there are (too) many possible search spaces which can be built: *deciding* at which dimension a dependence will be strongly solved has an impact on the search space constraints, and hence each possible decision leads to a potentially different search space. The second problem is that the constructed 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 solved for a given dimension. This solution is interesting because it reduces the number of dimensions and exhibits a parallelism that one may exploit, but the genuine greedy algorithm faces a lack of scalability due to the size of the integer programs to be resolved. Moreover, maximum reduction of the schedule dimension is likely to translate into big scheduling coefficients which are known to be a source of heavy control overhead when generating the target code for sequential targets.

We suggest a simple variation to overcome these issues. The following algorithm sketches our search space construction for a given SCoP:

1. Compute the exact set G of dependences for the SCoP by performing instancewise analysis [4]. Mark all dependences as weakly solved.
2. Starting at dimension $d = 1$, 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 $D \in G$:

- Express the space $\mathcal{T}_{\mathcal{D}}$ of legal schedules strongly respecting the causality condition 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 heuristic outputs for each schedule dimension d a space \mathcal{L}_d of legal solutions. 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 test⁴ for each dependence. Since in this study we address the problem of optimizing sequential codes, *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 Building Affine Multidimensional Schedules

The algorithm presented in Section 2.1 constructs a polytope per sequential schedule dimension, for a given program. Hence it is required to traverse these polytopes to build different legal program versions, and to provide efficient mechanisms for this traversal.

A program version is represented by its scheduling matrix Θ . First, are all schedule coefficients attached to iterators (i), for all statements. Second, are all schedule coefficients attached to parameters (p), for all statements. Third, are the schedule coefficient attached to 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 p statements and schedules of dimension s :

$$\Theta \cdot \mathbf{x} = \begin{pmatrix} i_1^1 \cdots i_p^1 p_1^1 \cdots p_p^1 c_1^1 \cdots c_p^1 \\ \vdots \\ i_1^s \cdots i_p^s p_1^s \cdots p_p^s c_1^s \cdots c_p^s \end{pmatrix} \cdot (\mathbf{x}_1 \cdots \mathbf{x}_p \mathbf{n}_1 \cdots \mathbf{n}_p 1 \cdots 1)^T$$

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

⁴ Over \mathcal{L}_d which contains exactly one variable per schedule coefficient

first shape the polytopes by computing their complete projection,⁵ 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. This correction procedure can be depicted as follows. Given a vector \mathbf{v} of size n , for $i \in [1, n]$, 1) compute the lower bound and upper bound of v_i , provided the values for $v_1 \cdots v_{i-1}$ and 2) 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 (for instance, pick values for the \mathbf{i} coefficients and set all other coefficients to 0); applying this correction principle on it will result in finding the minimal amount of complementary transformations needed to make this transformation sequence legal. In the following, we refer to this mechanism as the *completion algorithm*.

Three fundamental properties are embedded in this completion heuristic. Given a point \mathbf{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 \mathbf{p} coefficients (i.e. the minimal correction of the 0 value) translates into the maximal fusion available *for the picked \mathbf{i} coefficients*.

In practice this heuristic is extremely powerful: it allows to build only parts of the schedule, focusing on some of its properties, and the heuristic will automatically complete it with a minimal amount of correction to make it legal. It is intensively used to limit the number of transformation sequences tested, as shown in the following sections.

3 Performance Distribution

3.1 Experimental Protocol

For each point in the search space, we generate the corresponding C code with `ClooG` [2], add all the required instrumentation to the code, then compile it and run it on the target machine. Our target architecture is an AMD Athlon X64 3700+, 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. For every tested version (including the original code), we used the following optimization settings which are known to bring excellent performance for this platform: `-O3 -msse2 -ftree-vectorize`. The performance data are collected using hardware counters, thanks to 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 on 100 executions.

⁵ It implies the order to compute coefficient values is the reverse order of the variable projection, that is from \mathbf{i}_1 to c_p

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 which legality can be predicted using instancewise analysis. 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. Its statistics are summarized in Figure 3. For each schedule dimension, we report the degree of freedom (that is, the number of *different* schedules) decomposed in 3 different classes. The *i* class contains all the points with a distinct *i* prefix (that is, where iterator coefficients are going to be different, typically distinct legal combinations of interchange, skewing, reversal); then respectively for the *i + p* class (adding fusion, distribution); and the *i + 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 (it is the multiplication of the degree of freedom for each schedule dimension).

It is worth recalling that each program version corresponds to an arbitrarily complex sequence of transformations against the original program. It is possible to limit the degree of freedom to (a part of) the *i* or *i + p* classes, by simply

Schedule dimension	i	$i + p$	$i + p + c$
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`

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 various possibilities to make this sequence legal, but only one.

Performance distribution To limit the set of tested program versions, we rely on two empirical observations. Firstly, it is expected that traversing 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 $i + p$ class. Secondly, for the `dct` benchmark, it is expected that the third schedule dimension freedom 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.⁶ Eventually, we have to consider a search space of 1.29×10^6 different program versions, where each schedule coefficient which is not explored is computed with the completion algorithm.

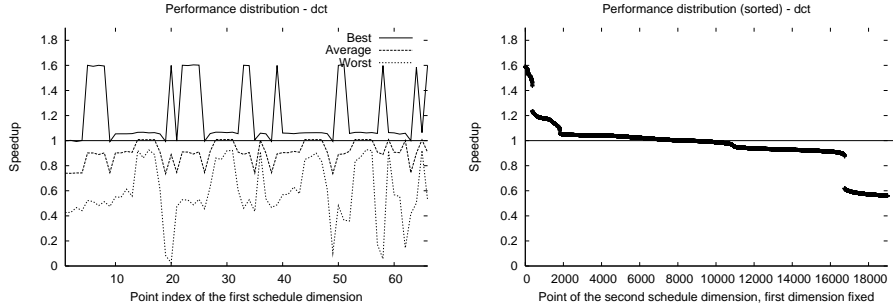
Figure 4 shows the performance distribution for the `dct` program. Figure 4(a) plots the *best*, *worst* and *average* performance for each point of the first schedule dimension Θ_1 (represented in the x axis). The *average* is computed from all the 19683 points in Θ_2 . The performance of the original 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 , provided the value of Θ_1 of the very best version.

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.

The difficulty to reach 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 it by a factor 2. Hence in this context it is expected that pure random approaches will fail to converge efficiently to the best speedup.

The performance distribution can be decoupled with respect to the schedule dimensions: there are several values for the first schedule dimension from which

⁶ We performed sampling also in the $i + p + c$ class as well as for the third schedule dimension: it always confirmed these assumptions.



(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`

it is impossible to attain the maximal performance. A contrario, the maximal performance is attainable from more than one point for the first dimension. We conclude that Θ_1 is a first discriminant of the performance distribution, nevertheless it lacks the ultimate precision one needs when designing search techniques.

Statistical analysis We achieve a more fine 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. We observe that 7 (out of 12) coefficients of the i class of Θ_1 have the same value (their variance is 0), as well as 2 (out of 5) coefficients of the p class. Also, 3 (out of 12) coefficients of the i class of Θ_2 have a very low variance, emphasizing that not only the first schedule dimension is necessary to characterize the distribution. The impact of these coefficients with the lowest variance on the complete distribution shape is then confirmed by correlating the performance of a version with non-optimal value for these coefficients: slight changes of any of the i coefficient of Θ_1 isolated by the variance computation translates into major performance variations, spanning its entire range.

Eventually, we observe that a relevant ordering of the impact of the several class of coefficients is $i < p < c$, and variance study confirmed our preliminary analysis that Θ_1 is the first discriminant of the performance distribution.

Hardware counters details Figure 5 gives more details on the performance source. It reports the behaviour for the *L1 accesses*, *L2 accesses* and *Branch*

count metrics. The performance of the original code is represented by a bold horizontal bar, each point below it improves the original code.

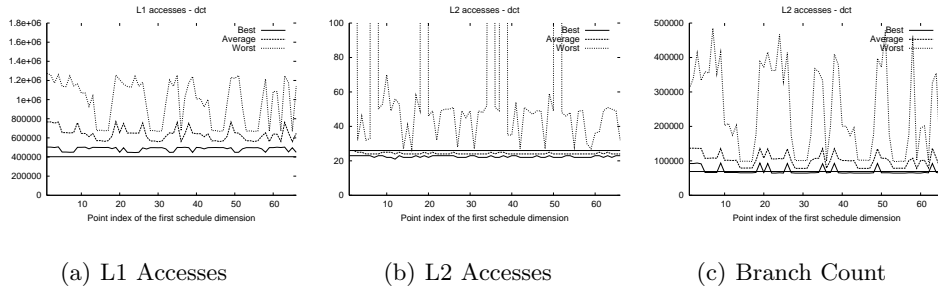


Fig. 5. Hardware Counters Distribution for *dct*

The most representative metric capturing the distribution shape is the *L1 accesses* curve. We observe that all transformations achieves a higher number of *L1 accesses* than the original code (at best 8% more). Conversely, several transformations achieve a lower number of *L2 accesses*. Hence the final memory criterion for optimal performance is minimization of both *L1* and *L2 accesses* in the space. The last reported indicator is Branch count. We can correlate this counter with the the program version controls: polyhedral code generation algorithms are likely to generated many complicated control statements (if and modulus) when applied highly complex transformations. While not 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 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 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. Overall, our results confirm the potential of iterative optimization to accurately capture the complex behavior of the processors and back-end compilers, it extends its applicability 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 some values from the *i* class and the dispersion of the performance distribution. In this section,

we study the influence of a strong limitation of the degree of freedom of the 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 i class of the first dimension.

Search space statistics on more examples In the following we focus on four 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, according to the same classes depicted in Figure 3. 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.	i	$i + p$	$i + p + c$
<code>latnrm</code>	11	75	3	1	9	27
<code>fir</code>	4	36	2	1	9	18
<code>lmsfir</code>	9	112	2	1	9	27
<code>iir</code>	8	66	3	1	9	18

Fig. 6. Search Space Statistics for Some UTDSP Benchmarks

We observe for each benchmark that the degree of freedom of the 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.

We show in the following how this nonexistent 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 $i + p$ class, for the first two schedule dimensions. Figure 7 shows the according performance distributions.

The first observation is again a significant speedup to discover: more than 30% speedup can be achieved for any of these benchmarks. Hence, for these benchmarks, the extremely limited degree of freedom of the first schedule dimension does not forbid to expose significant speedups.

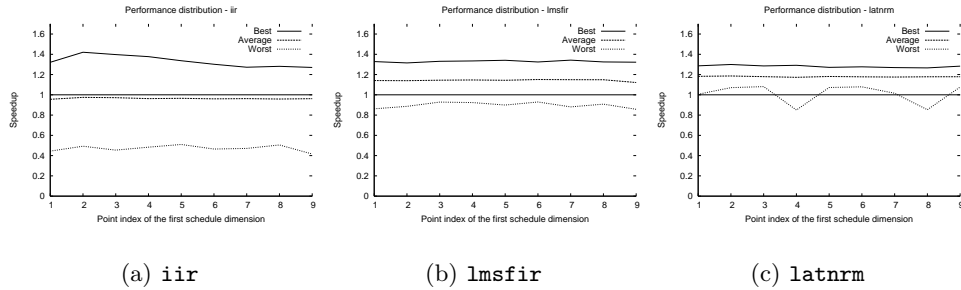


Fig. 7. Performance Distribution for 3 UTDSP benchmarks

As expected, the performance distribution is almost flat, another evidence of the impact of transformation coefficients: the degree of freedom in the i class translates into variations in the performance distribution.

3.4 Extension to Other Benchmarks

In order to generalize the results achieved for the `dct` benchmark, we need to distinguish two different concerns: the generalization to other programs, and the generalization to other architectures.

Extension to other programs We showed a positive correlation between the amount of variation in the performance distribution and the degree of freedom in the i class, especially for the first schedule dimension. Hence, in such situation, 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 $i + p$ 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.

Extension to other architectures Generalizing the results of the `dct` benchmark on AMD Athlon64 to other architectures must be done with care. Firstly, even if it is clear that the i class will still have a large impact on performance whatever the architecture, one has to pay attention to fusion and distribution: it is clearly transformations to focus on several embedded architectures. Hence again motivating the traversal of the degrees of freedom offered by the $i + p$ class. We also conducted experiments on the ST231 embedded VLIW processor, and even if extremely different of the AMD Athlon we still observed the exact same impact of the i class on the distribution shape. Although smaller exposed

speedups (the regular VLIW architecture is well captured in the STMicroelectronics compiler), our framework is still able to discover speedups on all tested benchmarks, with an average of 13.3%.

4 Efficient Search Space Traversal

From the extensive study of the performance distribution of several programs, we deduce the following:

1. The degree of freedom in the \mathbf{i} class of Θ_1 translates into variation in the performance distribution.
2. When the degree of freedom in the \mathbf{i} class of Θ_1 is nonexistent, the performance distribution is almost flat.
3. The impact of the schedule coefficients can be ordered: Θ_1 impacts more than Θ_2 , and inside a given schedule row, \mathbf{i} coefficients impacts more than \mathbf{p} and \mathbf{c} .

We leverage on these static 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.

4.1 The Traversal Heuristic

The traversal approach we propose relies on the evaluation of the degree of freedom available for the \mathbf{i} class of Θ_1 (referred to as $card(\mathcal{L}_1^{\mathbf{i}})$). Depending on its value, the heuristic will go more or less deeper in the set of available transformation sequences. In addition, we apply a feedback-driven filter. Experiments show that Θ_1 is an indicator of the overall performance distribution, hence we can start by traversing several values for Θ_1 with an unique value for the remaining Θ_k , and in a second step traversing several values for Θ_2 provided the best found value(s) for Θ_1 , etc. 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. $\Theta_k = completion(\mathcal{L}_k^{\mathbf{i}+\mathbf{p}+\mathbf{c}}), \forall k$
2. $d = 1$
3. if $card(\mathcal{L}_d^{\mathbf{i}+\mathbf{p}}) < L_1$, $Class = \mathbf{i} + \mathbf{p}$
else $Class = \mathbf{i}$
4. For all point p in \mathcal{L}_d^{Class}
 - (a) $\Theta_d = p . completion(\mathcal{L}_d^{\mathbf{i}+\mathbf{p}+\mathbf{c}-Class})$
 - (b) Evaluate Θ
5. Keep Θ_d for the best evaluated schedule(s) Θ , $d = d + 1$, go to 3

The parameter L_1 drives the exhaustiveness of the procedure: the larger the degree of freedom, the slower the convergence. By limiting to the \mathbf{i} class typically for inner schedule dimensions, we target only the most promising subspaces at

the expense of possibly missing the best transformation. 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 evaluations.

Figure 8 shows the results for `dct` together with seven kernels of the UTDSP suite amenable to polyhedral representation without code modification.⁷ We report the number of statements (`# Stm.`), the size of the `i` class for the first schedule dimension, the run index at which the best performing version was found (`Id Best`: the lower, the better), and the speedup obtained (`Speedup`).

	<code>dct</code>	<code>matmult</code>	<code>lpc</code>	<code>edge-c2d</code>	<code>iir</code>	<code>fir</code>	<code>lmsfir</code>	<code>latnrm</code>
<code>#Inst.</code>	5	2	12	3	8	4	9	11
<code>i</code>	39	76	243	1	1	1	1	1
<code>Id Best</code>	46	16	489	11	34	33	51	6
<code>Speedup</code>	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 far 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 a hardly predictable 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 in this case.

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%-100% 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 $\text{card}(\mathcal{L}_1^i) = 1$ to avoid traversing a space leaving few room for further improvements. This approach leads to an average 17.8% speedup on the 5 benchmarks where this criterion apply, without any evaluation.

⁷ `matmult` is a 2 statement matrix multiplication, for 10×10 matrices. See [10] for an extensive study of this kernel

4.2 Another Traversal Approach

When the degree of freedom available in the i class is high, our heuristic approach fails to converge in a reasonable time. This is because it relies at some level on the exhaustive traversal of some classes of coefficients. The probability for having a “too” large degree of freedom increases with the program size, and roughly programs of more than 10 statements need another traversal technique.

To overcome this limitation we introduced in our previous work a novel genetic algorithm traversal approach, tailored to the space properties (static and dynamic) [9]. In a word, we designed novel genetic operators that 1) preserve inclusion over an affinely constrained polytope, 2) encompass the relative importance of the several classes of coefficients in the probabilistic search 3) can be tuned for aggressiveness or refinement of the tested program versions. This approach allowed to discover significant speedups on programs up to 20 statements, on three representative architectures.

5 Conclusion

Model-based approaches to drive compiler optimizations fail to cover the complex interplay between hardware and compiler features. Iterative, empirical search techniques are now essential to cross this constantly growing “model-wall”. However, high level loop transformations challenge the design of such methods because, on one hand, *expressing* complex restructuring sequences is a difficult problem, and on the other hand, efficient methods to *traverse* huge and complex search spaces have to be designed.

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

To achieve efficient space traversing techniques, we present in this paper a detailed analysis of the performance distribution of affine multidimensional schedules. We build upon this extensive analysis a characterization of the distribution, mixing static and dynamic properties. From the static point of view, we show that a relative ordering of different classes of the search space variables can capture most of the distribution variations. We show how it dramatically helps to narrow the search to the most performance impacting ones. From the dynamic point of view, we show on a representative benchmark that the amount of optimizing versions discovering a significant fragment of the maximal speedup available is extremely low. We build upon these static and dynamic characteristics a powerful heuristic traversal method, exposing an average 32.56% speedup on 8 representative kernels, on AMD Athlon64 (single core).

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.