Companion paper for the 2nd HiPEAC Industrial Workshop

Iterative Optimization in the Polyhedral Model: One-Dimensional Scheduling case

Louis-Noël Pouchet, Cédric Bastoul and Albert Cohen

ALCHEMY, LRI - INRIA Futurs

firstname.lastname@inria.fr

Abstract

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

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

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

1. Introduction

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

- tuning or specializing an individual heuristic, adapting the profitability or decision model of a given transformation [31];
- tuning or specializing the selection and parameterization of existing (black-box) compiler phases [32, 1].

This paper takes a more offensive position in this fight. To avoid diminishing returns in tuning individual phases or combinations of those, we collapse multiple optimization phases into a single, unconventional, iterative search algorithm. By construction, the search space we explore encompasses *all legal program transformations* in a particular class. Technically, we consider the whole class of loop nest transformations that can be modeled as *one-dimensional schedules* [10], a significant leap in model and search space complexity compared to state-of-the-art applications of iterative optimization. We make the following contributions:

- we statically construct the optimization space of all, arbitrarily complex, arbitrarily long sequences of loop transformations that can be expressed as one-dimensional affine schedules (using a polyhedral abstraction);
- this search space is built free of illegal and redundant transformation sequences, avoiding them altogether at the very source of the exploration;
- we demonstrate multiple orders of magnitude reduction in the size of the search space, compared to filtering-based approaches on loop transformation sequences or state-of-the-art affine schedule enumeration techniques;
- these smaller search spaces are amenable to fast-converging, mathematically founded operation research algorithms, allowing to compute the exact size of the space and to traverse it exhaustively;
- our approach is compatible with acceleration techniques for feedback-directed optimization, in particular on machine-learning techniques which focus the search to a narrow set of most promising transformations;
- our source-to-source transformation tool yields significant performance gains on top of a heavily tuned, aggressive optimizing compiler.

Eventually, we were stunned by the intricacy of the transformed code, which was far beyond our expectations; a confirmation that whatever the performance model and whatever the expertise of the programmer, designing a predictive model for loop transformation sequences seems out of reach.

2. Related Work

Iterative compilation aims at selecting the best parameterization of the optimization chain, for a given program or for a given application domain. It typically affects optimization flags (switches), parameters (e.g., loop unrolling, tiling) and the phase ordering. [6, 5, 3, 20, 1, 25].

This paper studies a different search space: instead of relying on the compiler options to transform the program, we statically construct a set of candidate program versions, considering the distinct result of all legal transformations in a particular class. Our method is more tightly coupled with the compiler transformations and is thus complementary to other forms of iterative optimization. Furthermore, it is completely independent from the compiler back-end.

Because iterative compilation relies on multiple, costly "runs" (including compilation and execution), the current emphasis is on improving the profile cost of individual program versions [20, 13], or the total number of runs, using, e.g., genetic algorithms [19] or machine learning [31, 1]. Our meta-heuristic is tuned to the rich mathematical properties of the qunderlying *polyhedral* model of the search space, and exploits the regularity of this model to reduce the number of runs. Combining it with more generic machine

learning techniques seems promising and is the subject of our ongoing work.

The polyhedral model is a well studied, powerful mathematical framework to represent loop nests and to remove the main limitations of classical, syntactic loop transformations. Many studies have tried to assess a predictive model characterizing the best transformation within this model, mostly to express parallelism [21, 11] or to improve locality [35, 29, 24]. We present experimental results showing that such models, although associated with optimal strategies, fail to scratch the complexity of the target architecture and the interactions with the back-end compiler, yielding far from optimal results even on simple kernels.

Iterative compilation associated to the polyhedral model is not a very common combination. To the best of our knowledge, only Long et al. tried to define a search space based on this model [22, 23], using the Unified Transformation Framework [16] and targeting Java applications. Long's search space includes a potentially large number of redundant and/or illegal transformations, that need to be discarded after a legality check, and the fraction of distinct and legal transformations decreases exponentially to zero with the size of program to optimize. On the contrary, we show how to build and to take advantage of a search space which, by construction, contains no redundant and no illegal transformation.

3. Generating a Variety of Program Versions

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

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

3.1 An Algebraic Program Representation

Only parts of the program, called *Static Control Parts* (SCoP), can be represented algebraically in the polyhedral model. Roughly, a SCoP is a maximal set of consecutive instructions such that:

- the only allowed surrounding control structures are `for` loops and `if` conditionals,
- loop bounds and conditionals are affine functions of the surrounding loop iterators and the global parameters.

The importance of SCoPs has been widely discussed by Girbal et al. [14], showing that they capture a large portion of the computation time of scientific and signal processing applications.

In such a program class, semantic information can be represented as $\mathbb{Z}$-polyhedra. For instance, let us consider the `matvect` kernel in Figure 1.

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

Figure 1. `matvect` kernel

Instruction R is enclosed by a single loop iterating on i. Its *iteration vector* $\vec{x_R}$ is (i). Iterator i takes values between 0 and n, hence the polyhedron containing all the values successively taken by i is $\mathcal{D}_R : \{i \mid 0 \leq i \leq n\}$. Intuitively, to each point of the polyhedron corresponds an execution of instruction R, called an *instance*, where the value of the loop iterator i is the corresponding point coordinates in the polyhedron. With a similar reasoning we can express the iteration domain of instruction S: $\vec{x_S} = \binom{i}{j}$. The polyhedron representing its iteration domain is $\mathcal{D}_S = \{i, j \mid 0 \leq i \leq n \wedge 0 \leq j \leq n\}$.

In the remainder, we use a matrix representation with homogeneous coordinates to express systems of affine (not only linear) inequalities. For instance, for the iteration domain of R, we get :

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

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

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

Other kinds of array references have to be modeled conservatively. Pointers arithmetic is forbidden (except when translated by a former restructuring pass to array-based references [12]) and function calls have to be inlined.

3.2 One-Dimensional Schedules

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

A *one-dimensional* schedule, if it exists, expresses the program as a single *sequential* loop, possibly enclosing one or more *parallel* loops. A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops. Affine schedules have been extensively used to design systolic arrays [28] and in automatic parallelization programs [10, 8, 15], then have seen many other applications.

In this study, we focus on affine one-dimensional schedules: given a statement S, it is an affine form on the outer loop iterators

$\vec{x}_S$ and the global parameters $\vec{n}$. It is written

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

where T is a constant row matrix. Such a representation is much more expressive than sequences of primitive transformations, since a single one-dimensional schedule may represent a potentially intricate and long sequence of any of the transformations shown in Figure 2.

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

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

There exist robust and scalable algorithms and tools to reconstruct a loop nest program from a polyhedral representation (i.e., from a set of affine schedules) [17, 27, 4]. We will thus generate transformed versions of each SCoP by exploring its legal, distinct affine schedules, regenerating a loop nest program every time we need to profile its effective performance.

4. Building The Search Space

In general, restructuring a program will change its semantics. When a transformation preserves the original program semantics, we will say that it is *legal*. Previous works on iterative optimization using a polyhedral representation ensure this property by checking, after computing a transformation, whether it is legal or not [22, 23] (non-iterative optimization algorithms use either a similar approach [18], either consider programs simple enough that nearly every transformation is possible [36]). This results in considering huge search spaces, since every illegal or redundant solutions have to be checked, and to a significant computation overhead corresponding to each legality check (typically most of them stating that the transformation must not be applied). Such an approach cannot scale since the number of redundant and/or illegal transformations grows exponentially faster than the number of different and legal transformations with the size of the input program.

To overcome those issues, we propose to build a search space which, by construction, encompasses all legal program transformations in our one-dimensional schedule class. The following sections presents formally the search space construction, first by recalling how we can represent data dependences in our algebraic representation in section 4.1, then how we build the search space thanks to a deep result in linear algebra in section 4.2.

4.1 Data Dependence Representation

Two statements instances are in *dependence relation* if they access the same memory cell and at least one of these accesses is a write. For a program transformation to be correct, it is necessary to preserve the original execution order of such statement instances and thus to know precisely the instance pairs in dependence relation. In the algebraic program representation depicted in section 3.1, it is possible to characterize exactly the set of instances in dependence relation in a very synthetic way.

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

For instance, if we consider the `matvect` kernel in Figure 1, dependence analysis gives two dependence sets: instances of statement S depending on instances of statement R (R produces values used by S), $D_1 : R\delta S$, and similarly, $D_2 : S\delta S$. The dependence D_1 doesn't occur for each value of $\vec{x}_R$ and $\vec{x}_S$, but only if $i_R = i_S$. We can then define a *dependence polyhedron*, being a subset of the Cartesian product of the iteration domains, containing all the values of i_R, i_S and j_S for which the dependence exists. We can write this polyhedron in matrix representation (the first line represents the equality $i_R = i_S$, the two next ones the constraint that (i_R) have to belong to the iteration domain of R and similarly, the four last lines states that (i_S, j_S) belongs to the iteration domain of S):

$$\mathcal{D}_{D_1} : \left[\begin{array}{rrrrr} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{array}\right] \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \begin{array}{l} = 0 \\ \hline \geq \vec{0} \end{array}$$

4.2 Legal Transformation Space

The data dependence analysis gives the exact information on which statement instance pairs have to respect their relative original execution order. We can express the legality condition as an affine non-negative function over a given set of statement instances which are in dependence. Using the affine form of the Farkas Lemma [30] (a deep result in linear algebra), we can express the set of affine unique non-negative functions which meets the legality criterion. It will result in a linear system to solve for the dependence, where the solution set represents constraints on the schedule coefficients in order to ensure the respect of the precedence constraint imposed by the dependence.

For many dependence sets, the result is simply the intersection of the various constraint systems. Ultimately, we express in this way the space of all legal one-dimensional schedules for a given program of the SCoP class. Each point in this space corresponds to a valid, unique transformation sequence. To traverse this search space is now possible using polyhedra scanning methods [17, 27, 4].

5. Practical Search Space

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

have to ensure we remove only the less interesting solutions for performance.

It is done by using a side effect of the code generation algorithm: the code generated from schedules with high coefficients values is expected to embed too much complex controls. We so try to keep them as small as possible, with different values with regards to the type of the free variable attached to the coefficient.

We made several tests to compare our approach, taking into account only the legal schedules, to considering every schedules and filter legal ones thanks to a legality check, as Long et al. suggests [22]. We used different compute-intensive kernel benchmarks coming from various origins and listed in Figure 3. `h264` is a fractional sample interpolation of the H.264 standard [34]. `fir` and `fft` are DSP kernels extracted from UTDSP benchmark suite [34]. `lu`, `gauss`, `crout` and `matmul` are well known mathematical kernels corresponding to LU factorization, Gaussian elimination, Crout matrix decomposition and matrix-matrix multiply. `MVT` is a kernel including two matrix-vector multiplies, one matrix being the transposition of the other. `locality` is an hand-written memory access intensive kernel.

These kernels are typically small, from 2 to 17 statements. They are quite well adapted to the present study since first, they should not challenge present production compiler optimization schemes, and second, they will make it possible to achieve an exhaustive visit of our search space which is necessary to evaluate the potential of the method and to design heuristic techniques. Dealing with larger benchmarks presents some technical difficulties. First, every SCoP do not have a one-dimensional schedule and some preprocessing (such as using a single assignment form) may be necessary. We still not have tools to apply such preprocessing or even to extract useful program informations automatically (iteration domains, subscript functions, etc.) from the source code. The GRAPHITE framework inside GCC should soon provide such a facility and allow us to enlarge our benchmark set [26].

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

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

6. Scanning the Optimization Search Space

In previous sections, we formally defined how to build a singular search space where each point corresponds to a legal program version. We also adapted this space in such a way that a scan becomes possible in any case. In the following, we will actually visit the search space to evaluate its potential for program optimization. In section 6.1, we present our experimental setup, section 6.2 shows results on exhaustive search while section 6.6 presents a heuristic to avoid performing a large number of runs while preserving the core optimization benefits.

6.1 Experimental Setup

We implemented tools for dependence analysis, legal transformation space construction and scanning. We used for that purpose external publicly available tools as `PipLib`, a linear algebra tool [9] and `CLooG`, a code generator in the polyhedral model [4]. We designed our tools to be able to use them as a plugin in the future GRAPHITE GCC's polyhedral framework [26].

We ran our experiments on an Intel workstation based on Xeon 3.2GHz, 8KB L1, 512KB L2 caches. We used four different compilers: GCC 3.4.2, GCC 4.1.1, Intel ICC 9.0.1 and PathScale EKOPath 2.5. We used hardware counters to measure the number of cycles used by various programs. In order to avoid interferences with other programs and the system, we set the system scheduler policy to FIFO for every test. The kernel benchmark set is the one presented in section 5.

6.2 Exhaustive Space Scanning

Because our search space is only based on legal solutions, the acceptable number of solutions for our various kernel benchmarks makes it possible to achieve an exhaustive search in a reasonable amount of time. Figure 4 summarizes our results. The `Benchmark` column states the input original program; the `Compiler` column shows the compiler used to build each program version of the search space (GCC version was 4.1.1); the `Options` column precises the compiler options; the `Parameters` column gives the values of the global parameters (for instance the array sizes); the `Improved` column shows the number of transformations that achieves a better performance than the original program (the total number of versions is shown in Figure 3); the `ID best` gives the "number" of the best solution; lastly, the `Speedup` column gives the speedup achieved by the best solution with respect to the original program performance.

The two main results shown by this figure are, firstly, that the best program version highly depends on both compiler and compiler options. Even considering the several very best solutions, there are typically no intersection between the set of best transformations for two pairs compiler/compilation-options. Second, significant speedups are achieved thank to the traversal of the search space, demonstrating the interest of the method for optimizing compilation. In few cases, a 0% speedup is achieved, meaning that the original code was already optimal for our experimental setup. In average, the method leads to a 35.4% speedup, or to 14.9% without the extreme results of matrix-multiply kernel which is known to be a good candidate for such study.

Another interesting result is the form of the best transformed program since they typically appear to be quite complex. Most of the time, it was not possible to easily understand why they result in better performance since a significant part of the answer was related to the compiler design. We also noticed that optimization algorithm based on formal representations were sometimes far away from the optimal solution. A striking example is the `MVT` kernel benchmark. It is made of two matrix-vector multiplies in a perfectly nested i,j loop nest such that one of the matrices is the transposition of the other one. The simple, supposed optimal transformation in our class suggests a schedule of (i) for the first innermost statement and (j) for the second one, which results in accessing the same memory cell of the shared matrix during each iteration. The very best schedule was in fact $(j+1)$ and $(i+j+n+1)$ where n is the vector size, which result in distributing the statements in two different loop nests and to skew one of the resulting loop nests, leading to a weird data access pattern.

The relation with the compiler is described further in section 6.3. Section 6.4 deals with the effect of compiler options and lastly, we discuss the performance distribution in section 6.5.

Kernel Benchmark	#Dependences	$\vec{i}$-Bounds	$\vec{p}$-Bounds	c-Bounds	#Schedules	#Legal	Time
h264	15	$-1,1$	$-1,1$	$0,4$	3165	360	0.011
fir	12	$-1,1$	$-1,1$	$-1,1$	4.78×10^6	432	0.004
fft	36	$-2,2$	$-2,2$	$0,6$	5.8×10^{25}	804	0.079
lu	14	$0,1$	$0,1$	$0,1$	3.2×10^4	1280	0.005
gauss	18	$-1,1$	$-1,1$	$-1,1$	5.9×10^4	506	0.021
crout	26	$-3,3$	$-3,3$	$-3,3$	2.3×10^{14}	798	0.027
matmult	7	$-1,1$	$-1,1$	$-1,1$	19683	912	0.003
MVT	10	$-1,1$	$-1,1$	$-1,1$	4.7×10^6	16641	0.001
locality	2	$-1,1$	$-1,1$	$-1,1$	59049	6561	0.001

Figure 3. Search Space Computation

Benchmark	Compiler	Options	Parameters	#Improved	ID best	Speedup
h264	PathCC	-Ofast	none	11	352	36.1%
h264	GCC	-O2	none	19	234	13.3%
h264	GCC	-O3	none	26	250	25.0%
h264	ICC	-O2	none	27	290	12.9%
h264	ICC	-fast	none	0	N/A	0%
fir	PathCC	-Ofast	N=150000	240	72	6.0%
fir	GCC	-O2	N=150000	259	192	15.2%
fir	GCC	-O3	N=150000	119	289	13.2%
fir	ICC	-O2	N=150000	420	242	18.4%
fir	ICC	-fast	N=150000	315	392	3.4%
fft	PathCC	-O2	N=256 M=256 O=8	21	267	7.2%
fft	GCC	-O2	N=256 M=256 O=8	10	285	0.9%
fft	GCC	-O3	N=256 M=256 O=8	11	289	1.8%
fft	ICC	-O2	N=256 M=256 O=8	17	260	6.9%
fft	ICC	-fast	N=256 M=256 O=8	20	112	6.4%
lu	PathCC	-Ofast	N=1000	100	224	6.5%
lu	GCC	-O2	N=1000	321	339	1.6%
lu	GCC	-O3	N=1000	330	337	3.9%
lu	ICC	-O2	N=1000	281	770	9.0%
lu	ICC	-fast	N=1000	262	869	8.7%
gauss	PathCC	-Ofast	N=150	212	4	3.1%
gauss	GCC	-O2	N=150	204	2	1.7%
gauss	GCC	-O3	N=150	52	2	0.01%
gauss	ICC	-O2	N=150	63	288	0.05%
gauss	ICC	-fast	N=150	15	39	0.03%
crout	PathCC	-Ofast	N=150	0	N/A	0%
crout	GCC	-O2	N=150	132	638	3.6%
crout	GCC	-O3	N=150	56	628	1.7%
crout	ICC	-O2	N=150	37	625	0.5%
crout	ICC	-fast	N=150	63	628	2.9%
matmul	PathCC	-Ofast	N=250	402	283	308.1%
matmul	GCC	-O2	N=250	318	573	243.6%
matmul	GCC	-O3	N=250	345	143	248.7%
matmul	ICC	-O2	N=250	390	311	56.6%
matmul	ICC	-fast	N=250	318	641	645.4%
MVT	PathCC	-Ofast	N=2000	5652	4934	27.4%
MVT	GCC	-O2	N=2000	3526	13301	18.0%
MVT	GCC	-O3	N=2000	3601	13320	21.2%
MVT	ICC	-O2	N=2000	5826	14093	24.0%
MVT	ICC	-fast	N=2000	5966	4879	29.1%
locality	PathCC	-Ofast	N=10000, M=2000	6069	5430	47.7%
locality	GCC	-O2	N=10000, M=2000	30	5494	19.0%
locality	GCC	-O3	N=10000, M=2000	589	4332	6.0%
locality	ICC	-O2	N=10000, M=2000	3269	2956	38.4%
locality	ICC	-fast	N=10000, M=2000	4614	3039	54.3%

Figure 4. Search Space Statistics

6.3 The Compiler as an Element of the Target Platform

Our iterative optimization scheme is independent from the compiler and may be seen as a higher level to classical iterative compilation. In the same way as a given program transformation may better exploit a feature of a given processor, it also may enable more ag-

gressive options of a given compiler. Because production compilers have to generate a target code in any case in a reasonable amount of time, their optimizations are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase.

To study this behavior and estimating how a higher level iterative optimization scheme may lead to better performances, we achieved a exhaustive scan of our search space for various programs and compilers with aggressive optimization options. We illustrate our results in Figure 4, and with more details in Figure 6 for the matrix-multiply kernel shown in Figure 5, a very classic computational kernel. This kernel benchmark has been extensively studied, and is a typical target of aggressive optimizations of production compilers.

```
   for (i = 1; i <= n; i++)
   |  for (j = 1; j <= n; j++) {
S1 |  |  C[i][j] = 0;
   |  |  for (k = 1; k <= n; k++)
S2 |  |  |  C[i][j] = A[i][k] * B[k][j];
   |  }
```

Figure 5. `matmult`

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

Figure 6 shows significant speedups achieved by the best transformations for each back-end compiler. Such speedups are not uncommon when dealing with the matrix-multiplication kernel. The important point is that we do not perform any tiling (it requires multi-dimensional schedules), contrary to nearly all other works (see [36, 2] for useful references). It was possible to check using PathScale EKOPath that many optimization phases have been enabled or disabled, depending on the version generated from our exploration tool. Nevertheless it is technically hard to know precisely the contribution of the one-dimensional schedule (which has a high potential, by itself, as an optimizing transformation) with respect to the enabled compiler optimizations. But another striking result is the high variation of the best schedules depending on the compiler. For instance the lack of the j iterator in $\theta_{S1}(\vec{x}_{S1})$ for GCC or the lack of the n parameter $\theta_{S2}(\vec{x}_{S2})$ for ICC.

These results, which are consistent with the other tested programs, emphasize the need of a compiler-dedicated transformation to achieve the best possible performance. One possible explanation is the difference between optimization phases in the different back-end compilers. Compilers have attained such a level of complexity that it is no longer possible to model the effects of downstream phases on upstream ones. Yet it is mandatory to rely on the downstream phases of a back-end compiler to achieve a decent performance, especially those which cannot be embedded naturally in the polyhedral model.

6.4 On the Influence of Compiler Options

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

6.5 Discussion on Performance Distribution

Exhaustive scanning of all program versions is feasible on (small) kernels, and lets us observe the exact performance distribution. Figures 7 and 8 show this distribution for the `matmult`, `locality` and `crout` examples, considering multiple compilers and optimization options. Each graph represents the computation time of every point in the search space as a function of its number in the scanning order. An horizontal line shows the performance of the original program: every point below this line corresponds to a more efficient program version.

Although the scanning order may be a weird choice for such representation, it shows that the performance distribution is not totally chaotic.[1]

From these observations, we conclude that:

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

The impact of the compiler on the distribution is emphasized in Figure 8. Here we compare, for an identical original program (hence an identical optimization search space), the the distribution on `ICC -fast` and `GCC4 -O3` on the `crout` kernel benchmark. Hence, understanding performance regularities may help to find *hot* regions in the search space, thus avoiding useless runs in low-interest regions and diminishing-return searches among nearly optimal solutions. Machine learning techniques are used to solve similar problems for classical iterative optimization problems, and seem particularly promising to achieve this goal [31, 1]. We defer the application of these approaches to a further study, dedicating this paper to the study of the mathematical properties of our model, in an attempt to pruning the search space without loosing the most interesting solutions.

6.6 Heuristic Traversal of the Optimization Space

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

6.6.1 Decoupling Heuristic

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

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

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

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

Compiler	Option	Original	Best	Schedule	Speedup
GCC 3.4.2	-O3	519	163	$\theta_{S1}(\vec{x}_{S1}) = -1$ $\theta_{S2}(\vec{x}_{S2}) = k+1$	318.4%
GCC 4.1.1	-O3	515	207	$\theta_{S1}(\vec{x}_{S1}) = -i-j+n-1$ $\theta_{S2}(\vec{x}_{S2}) = k+n$	248.7%
ICC 9.0.1	-fast	465	72	$\theta_{S1}(\vec{x}_{S1}) = -i+n$ $\theta_{S2}(\vec{x}_{S2}) = k+1$	645%
PathCC 2.5	-Ofast	228	79	$\theta_{S1}(\vec{x}_{S1}) = j-n-1$ $\theta_{S2}(\vec{x}_{S2}) = k$	308%

Figure 6. Results for the `matmult` example

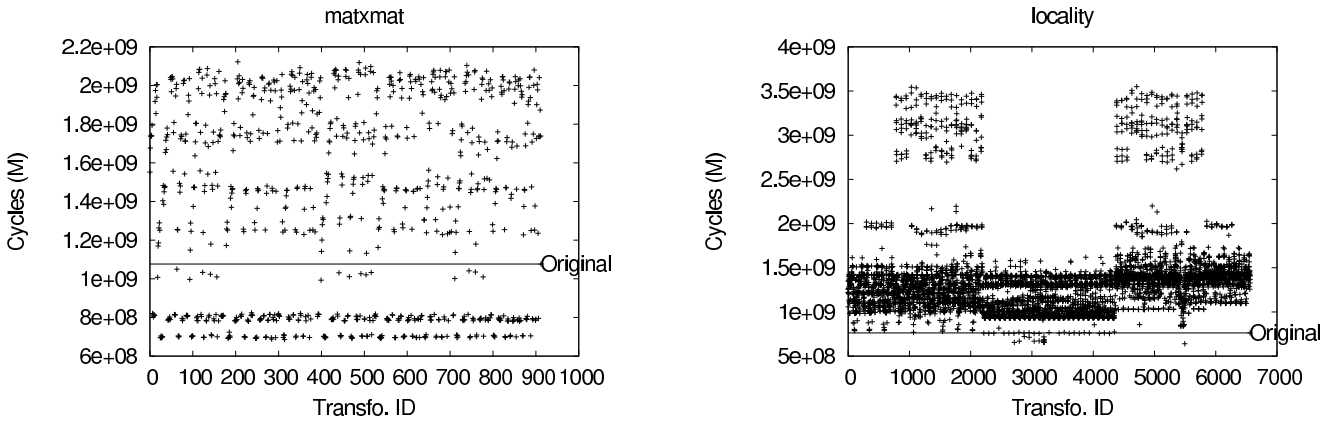

Figure 7. Performance Distributions for `matmult` and `locality` using `GCC4 -O2`

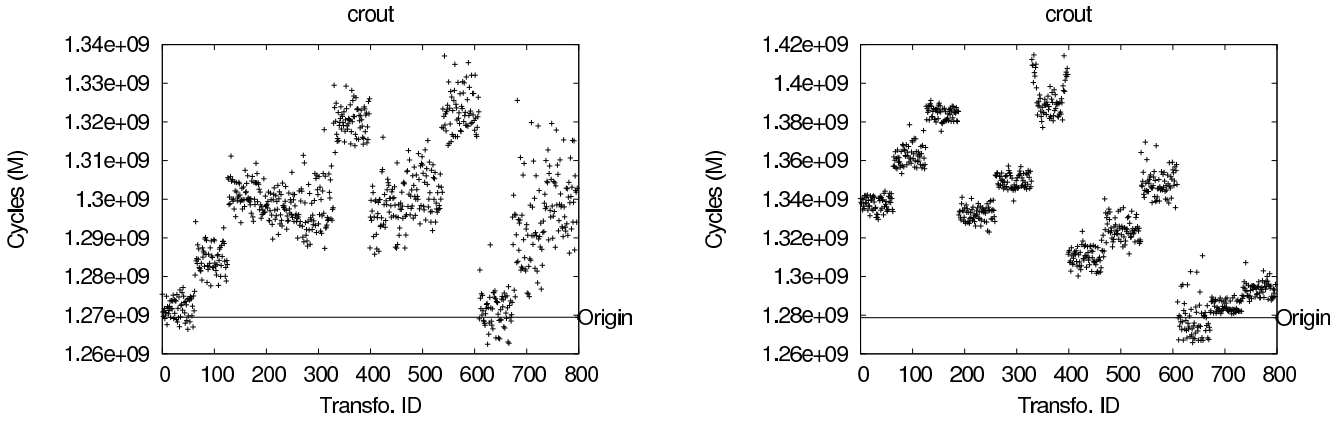

Figure 8. Performance Distributions for `crout, ICC -fast` and `crout, GCC4 -O3`

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

heuristic centered on the enumeration of the possible combinations for the $\vec{\imath}$ coefficients.

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

The heuristic can be sketched in 5 steps.

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

6.6.2 Discussion

Figure 9 details a run of our decoupling heuristic, and compares it witho a plain random search for some of our kernel benchmarks.[2] It shows the relative percentage of the best speedup achieved as a function of the number of iterative runs. A fast convergence of the decoupling heuristic is attained on some examples like `crout` or `locality`, where regularities in the distribution can be observed. On these tested examples, more than 95% of the maximum speedup can be achieved with an order of magnitude reduction of the number of runs, compared to an exhaustive scan.

On the other hand, we observed a suboptimal behavior of the heuristic comparatively to a full random driven approach, as Figure 9 shows for the `matmult` kernel. Not surprisingly, as soon as the density of interesting transformations is large, a random space scan may converge faster than our enumeration-based method.

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

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

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

[2] Some results are missing due to temporary instabilities in the experimental test-bed. We work to correct these bugs before the final version.

7. Future Work

Affine multidimensional schedules It is always possible to find a multidimensional affine schedule to a SCoP, while a one-dimensional schedule may not exists. Unfortunately, the generalization of our method to multidimensional schedules leads to a well known combinatorial barrier: if there is exactly one way to choose the set of dependences to satisfy in the one-dimensional case (they must all be satisfied in one dimension, i.e. in one set), there is a combinatorial way to choose the sets as soon as there is more than one dimension. Feautrier proposed a greedy algorithm to solve the maximal set of dependences at a given depth, and increment the depth if unresolved dependences remain [11]. This would give us the minimal sequential depth of the schedule [33], but the combinatorics remains if we want to explore all the legal schedules.

Parallelism The polyhedral model is designed to express in a natural way parallelism inside loop nests. Our study was only applied to monoprocessor machines, but it is a short term assignment to exploit this parallelism in a state-of-the-art shared-memory system. We need to slightly modify the code generation phase in order to generate an OpenMP equipped C code [7].

8. Conclusion

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

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

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

It is expected that a systematic exploration will not scale to large programs, or when multi-dimensional schedules are considered. But our study also contributes key observations about the performance distributions in the transformation space, a first step towards combining our search space construction and enumeration approach with more generic machine learning or empirical search techniques.

References

[1] AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 295–305.

[2] ALLEN, J., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] ALMAGOR, L., COOPER, K., GROSUL, A., HARVEY, T., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T.

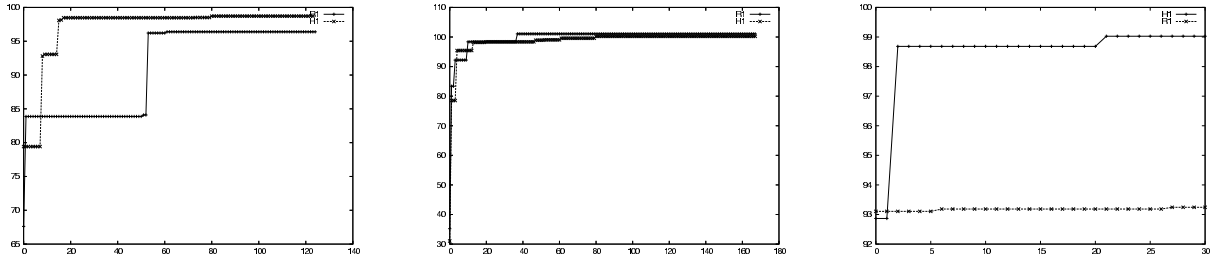

Figure 9. Comparison between the random and the decoupling heuristics, on the `locality`, `matmul` and `mvt` examples

Finding effective compilation sequences. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)* (New York, 2004), pp. 231–239.

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

[5] BODIN, F., KISUKI, T., KNIJNENBURG, P. M. W., O'BOYLE, M. F. P., AND ROHOU, E. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback Directed Compilation* (Paris, October 1998).

[6] CHOW, K., AND WU, Y. Feedback-directed selection and characterization of compiler optimizations. In *2nd Workshop on Feedback-Directed Optimization* (Israel, November 1999).

[7] DAGUM, L., AND MENON, R. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng. 5*, 1 (1998), 46–55. See http://www.openmp.org.

[8] DARTE, A., ROBERT, Y., AND VIVIEN, F. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

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

[10] FEAUTRIER, P. Some efficient solutions to the affine scheduling problem: one dimensional time. *International Journal of Parallel Programming 21*, 5 (october 1992), 313–348.

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

[12] FRANKE, B., AND O'BOYLE, M. Combining array recovery and high level transformation: an empirical evaluation for embedded processors. In *CPC'10 International Workshop on Compilers for Parallel Computers* (Amsterdam, January 1995), pp. 29–38.

[13] FURSIN, G., COHEN, A., O'BOYLE, M., AND TEMAM, O. A practical method for quickly evaluating program optimizations. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)* (Barcelona, Nov. 2005), no. 3793 in LNCS, Springer-Verlag, pp. 29–46.

[14] GIRBAL, S., VASILACHE, N., BASTOUL, C., COHEN, A., PARELLO, D., SIGLER, M., AND TEMAM, O. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming 34*, 3 (2006).

[15] GRIEBL, M., FABER, P., AND LENGAUER, C. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience 16*, 3 (march 2004), 221–246.

[16] KELLY, W., AND PUGH, W. A framework for unifying reordering transformations. Tech. rep., College Park, MD, USA, 1993.

[17] KELLY, W., PUGH, W., AND ROSSER, E. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation* (McLean, 1995).

[18] KODUKULA, I., AHMED, N., AND PINGALI, K. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (Las Vegas, june 1997), pp. 346–357.

[19] KULKARNI, P., ZHAO, W., MOON, H., CHO, K., WHALLEY, D., DAVIDSON, J., BAILEY, M., PAEK, Y., AND GALLIVAN, K. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (San Diego, California, USA, 2003), ACM Press, pp. 12–23.

[20] KULKARNI, P. A., HINES, S. R., WHALLEY, D. B., HISER, J. D., DAVIDSON, J. W., AND JONES, D. L. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim. 2*, 2 (2005), 165–198.

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

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

[23] LONG, S., AND FURSIN, G. Systematic search within an optimisation space based on unified transformation framework, 2006.

[24] MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4 (1996), 424–453.

[25] MONSIFROT, A., BODIN, F., AND QUINIOU, R. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications* (London, UK, 2002), Springer-Verlag, pp. 41–50.

[26] POP, S., COHEN, A., BASTOUL, C., GIRBAL, S., JOUVELOT, P., SILBER, G.-A., AND VASILACHE, N. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit* (Ottawa, Canada, June 2006).

[27] QUILLERÉ, F., RAJOPADHYE, S., AND WILDE, D. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming 28*, 5 (october 2000), 469–498.

[28] QUINTON, P., AND DONGEN, V. V. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing 1*, 2 (october 1989), 95–113.

[29] SCHREIBER, R., AND VILLARD, G. Lattice-based memory allocation. *IEEE Trans. Comput. 54*, 10 (2005), 1242–1257. Member-Alain Darte.

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

[31] STEPHENSON, M., AMARASINGHE, S., MARTIN, M., AND O'REILLY, U.-M. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not. 38*, 5 (2003), 77–90.

[32] TRIANTAFYLLIS, S., VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 204–215.

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

[34] WIEGAND, T., SULLIVAN, G., AND LUTHRA, A. Itu-t rec. h.264 – iso/iec 14496-10 avc - final draft. Tech. rep., Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, May 2003.

[35] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation* (New York, NY, USA, 1991), ACM Press, pp. 30–44.

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