Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model

Louis-Noel Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, R. Ramanujam, P. Sadayappan

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model

Louis-Noël Pouchet — Uday Bondhugula — Cédric Bastoul — Albert Cohen — J. Ramanujam — P. Sadayappan

N° 6962

June 2009

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--6962--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model

Louis-Noël Pouchet *, Uday Bondhugula† , Cédric Bastoul* ,
Albert Cohen* , J. Ramanujam‡ , P. Sadayappan§

Domaine 2 — Algorithmique, programmation, logiciels et architectures
Équipes-Projets ALCHEMY

Rapport de recherche n° 6962 — June 2009 — 26 pages

Abstract: On modern architectures, a missed optimization can translate into performance degradations reaching orders of magnitude. More than ever, translating Moore's law into actual performance improvements depends on the effectiveness of the compiler. Moreover, missing an optimization and putting the blame on the programmer is not a viable strategy: we must strive for portability of performance or the majority of the software industry will see no benefit in future many-core processors.

As a consequence, an optimizing compiler must also be a parallelizing one; it must take care of the memory hierarchy and of (re)partitioning computation to best suit the target architecture. Polyhedral compilation is a program optimization and parallelization framework capable of expressing extremely complex transformation sequences. The ability to build and traverse a tractable search space of such transformations remains challenging, and existing model-based heuristics can easily be beaten in identifying profitable parallelism/locality trade-offs. *We propose a hybrid iterative and model-driven algorithm for automatic tiling, fusion, distribution and parallelization of programs in the polyhedral model*. Our experiments demonstrate the effectiveness of this approach, both in obtaining solid performance improvements over existing auto-parallelizing compilers, and in achieving portability of performance on various modern multi-core architectures.

Keywords: Affine Scheduling, Tiling, Iterative Compilation, Loop Transformation,

* ALCHEMY, Inria Saclay Île-de-France and University of Paris Sud 11
† IBM T.J. Watson Research, Yorktown Heights, New York
‡ Louisiana State University
§ The Ohio State University

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Contents

1 Introduction

Portability of performance over a broad range of architectures is a difficult task. It requires: (1) to be able to apply complex sequences of loop transformations; and (2) to effectively model the interplay of all hardware components involved in program execution. Current model-driven optimization heuristics either rely on a restricted subset of the possible program transformations, or simply fail to be portable.

The polyhedral representation of programs allows expression of arbitrarily complex sequences of loop transformations. The downside of this expressiveness is the extreme difficulty in selecting a good optimization strategy combining the most important loop transformations, including loop tiling, fusion, distribution, interchange, skewing, permutation and shifting [14, 24]. It is also hard for any model-based technique to capture analytically the complex interplay between hardware components, taking into account downstream optimization passes.

Considering the state-of-the-art tiling and parallelization algorithm in the polyhedral model [6], we show that a purely model-based approach fails to achieve performance portability. It is not sufficient since several other high-impact factors, including cache conflicts and vectorization, are not taken into account. Tuning a transformation to best fit the architectural constraints is required, and we wish to exhibit portable performance, trading scalability, locality and synchronization overhead efficiently for any target.

To address these challenges, we designed a hybrid scheme for optimization and parallelization. It relies on an iterative, feedback-directed exploration of loop fusion / distribution choices, that efficiently drives a model-based algorithm for many other loop transformations, including loop tiling. Portability is achieved thanks to iteratively testing for different program versions, and our method finds the optimal version on the benchmarks we considered. We obtained improvements ranging from $1.1\times$ to $21.3\times$ over reference parallelizing compilers using model-based heuristics, and validated the portability of our approach considering three different modern multi-core architectures (Intel Dunnington, AMD Shanghai and IBM Power5+).

The rest of the paper is organized as follows. Section 2 recalls the fundamental concepts and presents the model-based optimization algorithms used in our hybrid strategy. Section 3 details the search space construction, pruning and traversal strategy. Section 4 evaluates this technique experimentally, and Section 5 discusses related work.

2 Automatic Parallelization and Optimization

Most compiler internal representations match the inductive semantics of imperative programs (syntax tree, call tree, control-flow graph, SSA). In such reduced representations of the dynamic execution trace, a statement of a high-level program occurs only once, even if it is executed many times (e.g., when enclosed within a loop). Representing a program in this manner is not convenient for optimizations that need a representation granularity at the level of dynamic *statement instances*. For example, transformations like loop interchange, fusion or tiling operate on the execution order of statement instances [34]. In addition, a rich algebraic structure is required when building complex compositions of such transformations [14], enabling efficient search space construction and traversal heuristics [24].

2.1 Polyhedral Model

The *polyhedral model* is a flexible and expressive representation for loop nests with statically predictable control flow. Loop nests amenable to algebraic representation are called *static control parts* (SCoP) [11, 14], roughly defined as a set of consecutive statements such that loop bounds and conditionals involved are affine functions of the surrounding loop iterators and global variables (constant that are unknown at compile time). Relaxation of these constraints based on affine over-approximations have been proposed [3]. Our optimization scheme is compatible with it, but we limit this work to describing the representation and optimization of regular SCoPs.

2.1.1 Representing Programs

Polyhedral program optimization is a three stage process. First, the program is analyzed to extract its polyhedral representation, including dependence information and access pattern.

The set of all executed instances of each statement is called an *iteration domain*. These sets are represented by affine inequalities involving the loop iterators and the global variables. Considering the `gemver` kernel in Figure 1, the iteration domain of $S1$ is:

$$\mathcal{D}_{S1} = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i < \mathrm{M} \wedge 0 \leq j < \mathrm{M}\}$$

$\mathcal{D}_{S1}$ is a (parametric) integer polyhedron, that is a subset of $\mathbb{Z}^2$. The *iteration vector* $\vec{x}_{S1}$ is the vector of the surrounding loop iterators, for $S1$ it is (i,j) and takes value in $\mathcal{D}_{S1}$.

```
    for(i=0; i<M; i++)
      for(j=0; j<M; j++)
S1      A[i][j] = A[i][j] + u1[i] * v1[j]
                  + u2[i] * v2[j];
    for(i=0; i<M; i++)
      for (j=0; j<M; j++)
S2      x[i] = x[i] + beta*A[j][i]*y[j];
    for(i=0; i<M; i++)
S3    x[i] = x[i] + z[i];
    for(i=0; i<M; i++)
      for(j=0; j<M; j++)
S4      w[i] = w[i] + alpha*A[i][j]*x[j];
```

Figure 1: `gemver` original code

Two statements instances are in *dependence relation* if they access the same memory cell and at least one of these accesses is a write. Given two statements R and S, a *dependence polyhedron* $\mathcal{D}_{R,S}$ is a subset of the Cartesian product of $\mathcal{D}_R$ and $\mathcal{D}_S$. $\mathcal{D}_{R,S}$ contains all pairs of instances $\langle \vec{x}_R, \vec{x}_S \rangle$ such that $\vec{x}_S$ depends on $\vec{x}_R$, for a given array reference. Hence, for an optimization to respect the program semantics, it must ensure that $\vec{x}_R$ is executed before $\vec{x}_S$, for all pairs $\langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}$.

To capture all program dependences we build a set of dependence polyhedra, one for each pair of array references accessing the same array cell (scalars being a particular

case of array), thus possibly building several dependence polyhedra per pair of statements. The *polyhedral dependence graph* is a multi-graph with one node per statement, and an edge $e^{R \to S}$ is labeled with a dependence polyhedron $\mathcal{D}_{R,S}$, for all dependence polyhedra.

2.1.2 Representing Optimizations

The second step of polyhedral program optimization is to pick a transformation for the program. Such a transformation captures in a single step what may typically correspond to a sequence of several tens of textbook loop transformations [14]. It takes the form of a carefully crafted affine multidimensional schedule, together with (optional) iteration domain or array subscript transformations.

In this work, a given loop nest optimization is defined by a multidimensional affine schedule. Given a statement S, we use an affine form on the outer loop iterators $\vec{x}_S$. It is written

$$\Phi^S(\vec{x}_S) = C_S \begin{pmatrix} \vec{x}_S \\ 1 \end{pmatrix}$$

where C_S is a matrix of non-negative integer constants. Multidimensional dates can be seen as logical clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes, and so on. Note that every static control program has a multidimensional affine schedule [12], and that any loop transformation can be represented in the polyhedral representation [34].

We note ϕ_i^S the i^{th} row of C_S. A row is an *affine hyperplane* on $\mathcal{D}_S$. For S with m_S outer loop iterators we note:

$$\phi_i^S = [\, c_1^S \; c_2^S \; \dots \; c_{m_S}^S \; c_0^S \,]$$

That is, c_0^S is the coefficient attached to the scalar.

Multidimensional polyhedral tiling is applied by modifying the iteration domain of the statements to be tiled, in conjunction with further modifications of Φ [14].

Finally, syntactic code is generated back from the polyhedral representation on which the optimization has been applied. We use the state-of-the art code generator CLooG [2] to perform this task.

2.2 Hybrid Optimization

We organize the optimization and automatic parallelization of a loop nest as an iterative, feedback-directed search. Each iteration of the search is further decomposed into two stages:

1. choose a partition of the program statements, such that statements inside a given class can share at least one common loop in the generated code;
2. on each class of this partition, apply a model-driven, tiling-based optimization and parallelization algorithm.

2.2.1 Fusion/Distribution Strategy

The first step of our optimization process is to partition the statements of the program, such that a class of this partition corresponds to a set of *fusable statements*. The partition of all statements is called a *fusion structure* for the program: each statement is given the identifier p_{id} of the class it belongs to.

The class identifier reflects the distribution of statements across outer loops of the generated code. So a fusion structure is a *total preorder*[1] on the n statements of a program. For instance, the partition $fs(S1,S2,S3,S4) = (0,0,1,2)$ of `gemver` corresponds to the code in Figure 2. Intuitively, $S1$ and $S2$ are executed at the same "day" (under the same outer loop), then $S3$ in the next outer loop, then $S4$ in the last one.

```
    for(i=0; i<M; i++)
      for(j=0; j<M; j++) {
S1      A[i][j] = A[i][j] + u1[i] * v1[j]
                  + u2[i] * v2[j];
S2      x[j] = x[j] + beta * A[i][j] * y[i];
      }
    for(i=0; i<M; i++)
S3    x[i] = x[i] + z[i];
    for(i=0; i<M; i++)
      for(j=0; j<M; j++)
S4      w[i] = w[i] + alpha * A[i][j] * x[j];
```

Figure 2: `gemver` fusion structure $(0,0,1,2)$

It is very hard to predict a profitable fusion structure, due to the combinatorial of the problem and to the very complex and chaotic interplay of transformations resulting from the selection of a given partition. To find a profitable (target-specific) fusion structure for a given loop nest, we will thus resort to iterative, feedback-directed search.

To make sure that statements in a given class are fusable without impacting the program semantics, we leverage the polyhedral representation and define fusability as follows.

Definition 1 (Fusability of two statements) *Given two statements R and S, they are fusable (through a fine-grain interleaving) at dimension d if, for all dimensions* 1 *to d and for all dependence polyhedra* $\mathcal{D}_{R,S}$*,*

$$\begin{gathered}\phi_d^S(\vec{x}_S) - \phi_d^R(\vec{x}_R) \;\geq\; 0, \quad \langle x_R, x_S\rangle \in \mathcal{D}_{R,S} \\ \sum_{i=1}^{m_R} c_i^R > 0,\ \sum_{i=1}^{m_S} c_i^S > 0\end{gathered} \tag{1}$$

This definition takes into account any composition of loop interchange, skewing, shifting and peeling that is required to fuse the statements under a common loop (possibly with prologue and epilogue). Notice that, strictly speaking, it is possible to find ϕ^R, ϕ^S which respects this definition and still result in distributed statements, for instance when the loop bounds are scalars and c_0 is greater than the loop bound, thus representing a shift larger than the iteration domain, that is, a distribution. To avoid this case, loops are represented with a fictitious parameter for the upper bound when they have a scalar upper bound, preventing us from finding a value for c_0 larger than the loop bound. The constraints on ϕ^R, ϕ^S coefficients guarantee that some instances share the same time-stamp at level 1 to d; these instances are thus effectively fused.

[1] A total preorder is defined as a relation which is reflexive, transitive and total.

To model distribution, we insert in Φ, for each loop level k to distribute, a row ϕ_k such that $\phi_k = p_{id}$. So given R and S such that $p^R_{id} \neq p^S_{id}$, all timestamps at dimensions k and greater will be distinct and hence statements cannot be fused starting level k.

2.2.2 Tiling Hyperplanes

The second step is to apply, individually on each class of the partition, a polyhedral transformation which combines complex compositions of multi-dimensional tiling, fusion, skewing, interchange, shifting, and peeling, known as the Tiling Hyperplanes method [5, 6].

Tiling (or blocking) is a crucial loop transformation for parallelism and locality. The tiling hyperplane method computes an affine multidimensional schedule such that parallel loops are brought to the outer levels, and loops with dependences are pushed inside [5, 6]; at the same time, the number of dimensions that can be tiled are maximized. This technique is applied locally on each class, hence maximizing parallelism at the class level without disturbing the outer level fusion structure.

Tiling along a set of dimensions is legal if it is legal to proceed in fixed block sizes along those dimensions: this requires dependences to not be backward along those dimensions, thus avoiding a dependence path going out of and coming back into a tile; this makes it legal to execute the tile atomically. (1) must hold true for all dimensions to be tiled [16, 27, 5]. In addition, a second constraint is used to enforce linear independence: hyperplanes obtained (rows of Φ^S) level-by-level are linearly independent.

The algorithm proceeds by computing the schedule level by level, from the outermost to the innermost. At each level, a set of legal hyperplanes is computed for the considered statements, according to the cost model defined in Section 2.2.3. Dependences satisfied by these hyperplanes are removed, and another set is computed for the next level such that the new set is independent to all previously computed sets, and so on until all dependences have been satisfied. If at a given level it is not possible to find legal hyperplanes for all statements, the statements are split [5], resulting in a loop distribution at this level.

2.2.3 Static Cost Model

There are infinitely many hyperplanes that may satisfy the legality criterion (1). An approach that has proved to be simple, practical, and powerful has been to find those directions that have the shortest dependence components along them [5]. For polyhedral code, the distance between dependent iterations can always be bounded by an affine function of the global parameters, represented as a p-dimensional vector $\vec{n}$.

$$\mathbf{u}.\vec{n} + w \geq \phi^S(\vec{x}_S) - \phi^R(\vec{x}_R) \quad \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S} \qquad \mathbf{u} \in \mathbb{N}^p, w \in \mathbb{N} \tag{2}$$

The legality and bounding function constraints from (1) and (2) are recast through the affine form of the Farkas Lemma [12] such that the only unknowns left are the coefficients of ϕ and those of the bounding function, namely $\mathbf{u}$, w. Coordinates of the bounding function are then used as the minimization objective to obtain the unknown coefficients of ϕ.

$$\text{minimize}_{\prec} \ (\mathbf{u}, w, \ldots, c_i, \ldots) \tag{3}$$

The resulting transformation is a complex composition of multidimensional loop fusion, distribution, interchange, skewing, shifting and peeling. For each class in a

partition, several goals are achieved through this cost model: (1) maximizing coarse-grained parallelism, (2) minimizing communication and frequency of synchronization, and (3) maximizing locality. Finally, multidimensional tiling is applied on all permutable bands. Tile sizes are computed such that data accessed by each tile roughly fits in the L1 cache.

3 Building Program Versions

This section returns to the problem of selecting a profitable partition for the set of program statements. Each chosen fusion structure corresponds to a different transformed program version. We wish to find the best performing one, trading scalability, locality and synchronization overhead efficiently for any program and target.

With our approach, this trade-off has several consequences as fusion also impacts tilability and parallelism. Firstly, maximally fusing statements may hinder parallelization and vectorization, and there is a trade-off between improving locality and increasing parallelization possibilities. Secondly, fusion may interfere with hardware prefetching. Also, after fusion, too many data spaces end up using the same cache, reducing the effective cache capacity of each statement. Conflict misses are also likely to increase. Obviously, systematically distributing loops is a no better solution as it may be detrimental to locality.

Our approach consists of building a search space of candidate fusion structures, and for each of them, we then perform a full optimization thanks to the method presented in the previous section.

3.1 Properties of the Search Space

The search space is modeled as a convex set of candidate fusion structures, defined by affine inequalities. There are several motivating factors. The set of possible fusion structures of a program is extremely large (in the order of 10^{12} possibilities for 14 elements [29], with a super-exponential growth), while the space complexity of a carefully designed convex set hardly depends on the cardinality of the represented set. Also, removing a subset of unwanted fusion structures is made tractable as it involves adding affine constraint(s) to the space, in contrast to other representations that would require enumerating all elements for elimination. Finally, the issue of efficiently scanning a search space represented as a well-formed polytope has been addressed [25, 24], and these techniques apply directly.

Previous research on building a convex search space of legal affine schedules highlighted the benefits of integrating the legality criterion directly into the search space, leading to orders of magnitude smaller search spaces [25, 24]. This is critical to allow any iterative search method to focus on relevant candidates only. The following two properties are mandatory to achieve scalability, even on the smallest kernels:

1. there are no duplicates in the space, each candidate in the set is distinct
2. there are only legal candidates in the space, all preserve original program semantics.

A significant difference between building a set of affine schedules [25] and building the set of legal fusion structures is the definition of a duplicate. An affine schedule is duplicate if and only if all its coefficients are the same as that of another schedule,

in essence due to the generated code being syntactically different from another case. When considering the problem of fusion structures, we are only interested in representing distinct *relative* orderings of statements. For instance, given a program with two statements R and S, the fusion structures $fs(R,S) = (0,0)$ and $fs(R,S) = (1,1)$ are duplicates: they both represent the statements being fused, and only one of the two must be in the search space. This motivates the need for a stronger convex representation of distinct relative orderings, known as total preorders.

3.2 A Convex Space of All, Distinct Total Preorders

Given a set $\mathcal{S}$ with n elements, we define $\mathcal{P}$ as the set of all and distinct total preorders of its n elements. Let O be the target convex set we are looking for. The key problem is to guarantee one-to-one correspondence between points in O and $\mathcal{P}$, while preserving the convexity of O. Note that this problem is not purely reminiscent of standard order theory problems: we look for the *set* of *all distinct* total preorders of n elements, in contrast of previous work defining counting functions of this set [29].

To the best of our knowledge, uniqueness cannot be modeled in a convex fashion on a set with n variables. We propose to model an order between two elements $i,\ j \in \mathcal{S}$ by using three *binary* decision variables, defined as follows. $p_{i,j} = 1$ if and only if i precedes j, $e_{i,j} = 1$ if and only iff i equals j and $s_{i,j} = 1$ if and only if i succeeds j. To model the entire set, we introduce three binary variables for each non-ordered pair of elements, i.e., all pairs (i,j) such that $1 \leq i < n,\ i < j \leq n$. They are gathered in the set, O, containing $3 \times n(n-1)/2$ variables.

$$O = \left\{ \begin{array}{l} 0 \leq p_{i,j} \leq 1 \\ 0 \leq e_{i,j} \leq 1 \\ 0 \leq s_{i,j} \leq 1 \end{array} \right\}$$

For instance, the fusion structure $fs(S1,S2,S3,S4) = (0,0,1,2)$ of Figure 2 is represented by:

$$e_{1,2} = 1,\ e_{1,3} = 0,\ e_{1,4} = 0,\ e_{2,3} = 0,\ e_{2,4} = 0,\ e_{3,4} = 0$$
$$p_{1,2} = 0,\ p_{1,3} = 1,\ p_{1,4} = 1,\ p_{2,3} = 1,\ p_{2,4} = 1,\ p_{3,4} = 1$$

with all $s_{i,j}$ coefficients set to 0. Then, one can easily recompute the corresponding total preorder $(0,0,1,2)$, for instance by taking the lexicographic minimum of a system $\mathcal{W}$ of 4 non-negative variables that embed the order constraints modeled by all $p_{i,j}$, $e_{i,j}$ and $s_{i,j}$ values.

A key issue is the consistency of orderings: we need to guarantee that a total preorder can be recomputed from *any* point in O. An inconsistent ordering occurs when $\mathcal{W}$ is infeasible. Without any further conditions, inconsistent orderings may be generated, for instance by setting $e_{1,2} = 1$ and $p_{1,2} = 1$, a contradiction. As a first condition for consistency, we have, for all decision variables, the following mutual exclusion equality:

$$p_{i,j} + e_{i,j} + s_{i,j} = 1 \tag{4}$$

For the sake of simplicity, we immediately get rid of the $s_{i,j}$ variables, thanks to (4). We also relax (4) to get:

$$p_{i,j} + e_{i,j} \leq 1$$

Summing up, the general form of the set is:

$$O = \left\{ \begin{array}{c} 0 \leq p_{i,j} \leq 1 \\ 0 \leq e_{i,j} \leq 1 \\ p_{i,j} + e_{i,j} \leq 1 \end{array} \right\}$$

Mutually exclusive decision variables describe the order of a single pair of elements, and so each point in O that describes a consistent ordering is unique. However, some points in O still do not represent a total preorder, and they have to be removed.

Let us observe that inconsistent orderings in O have a single origin. They arise from impossible values for some variables *due to the value of other variables*. In other words, they arise from missing transitive conditions in picking a value of a given coefficient with respect to the values of other coefficients. We have to guarantee that all points in O preserve the totality and the transitivity conditions that a total preorder relation satisfies.

Basic transitivity of e coefficients For instance, to respect the transitivity of the relation, the following rule must hold true for all points in O. For some k:

$$e_{i,j} = 1 \wedge e_{i,k} = 1 \Rightarrow e_{j,k} = 1$$

We will omit the $= 1$ in the rest of the paper. Then, the following equation is equivalent to Eq (3.2):

$$e_{i,j} \wedge e_{i,k} \Rightarrow e_{j,k}$$

Similarly, we have:

$$e_{i,j} \wedge e_{j,k} \Rightarrow e_{i,k}$$

These two rules set the basic transitivity of e variables. These conditions are easily modeled as affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n], & e_{i,j} + e_{i,k} \leq 1 + e_{j,k} \\ & e_{i,j} + e_{j,k} \leq 1 + e_{i,k} \end{array} \right\}$$

By applying a similar reasoning, we can collect all constraints to enforce the transitivity of the total preorder relation as shown in the following.

Basic transitivity of p coefficients We apply a similar reasoning for the p coefficients. We have:

$$p_{i,k} \wedge p_{k,j} \Rightarrow p_{i,j}$$

This translates into:

$$\left\{ \begin{array}{ll} \forall k \in]i,j[, & p_{i,k} + p_{k,j} \leq 1 + p_{i,j} \end{array} \right\} \quad (5)$$

Complex transitivity on p and t coefficients We also have transitivity conditions imposed by a connection between the value for some e coefficients and some p ones. For instance, $R < S$ and $S = T$ implies $R < T$. The general equations for those cases are:

$$e_{i,j} \wedge p_{i,k} \Rightarrow p_{j,k}$$

$$e_{i,j} \wedge p_{j,k} \Rightarrow p_{i,k}$$

$$e_{k,j} \wedge p_{i,k} \Rightarrow p_{i,j}$$

These translate to the following affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n] & e_{i,j} + p_{i,k} \leq 1 + p_{j,k} \\ & e_{i,j} + p_{j,k} \leq 1 + p_{i,k} \\ \forall k \in]i,j[& e_{k,j} + p_{i,k} \leq 1 + p_{i,j} \end{array} \right\} \tag{6}$$

Complex transitivity on s and p coefficients Lastly, we have to take into account the transitivity on the fictitious s variables (those modeling $R > S$). The transitivity condition is:

$$s_{i,k} \wedge p_{j,k} \Rightarrow s_{i,j}$$

Since the reduction equation gives:

$$s_{i,j} = 1 - p_{i,j} - e_{i,j}$$

with $p_{i,j}$ and $e_{i,j}$ being mutually exclusive, the rule translates to the following affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n] & e_{i,j} + p_{i,j} + p_{j,k} \leq 1 + p_{i,k} + e_{i,k} \end{array} \right\} \tag{7}$$

General formulation of O All the previous constraints are gathered in the following expression of O, the convex set of all, distinct total preorders of n elements. For $1 \leq i < n,\ i < j \leq n,\ O$ is:

$$\left\{ \begin{array}{lll} & 0 \leq p_{i,j} \leq 1 \\ & 0 \leq e_{i,j} \leq 1 \end{array} \right\} \text{Variables are binary}$$

$$\left\{ \begin{array}{lll} & p_{i,j} + e_{i,j} \leq 1 \end{array} \right\} \text{Relaxed mutual exclusion}$$

$$\left\{ \begin{array}{lll} \forall k \in]j,n] & e_{i,j} + e_{i,k} \leq 1 + e_{j,k} \\ & e_{i,j} + e_{j,k} \leq 1 + e_{i,k} \end{array} \right\} \text{Basic transitivity on } e$$

$$\left\{ \begin{array}{lll} \forall k \in]i,j[& p_{i,k} + p_{k,j} \leq 1 + p_{i,j} \end{array} \right\} \text{Basic transitivity on } p$$

$$\left\{ \begin{array}{lll} \forall k \in]j,n] & e_{i,j} + p_{i,k} \leq 1 + p_{j,k} \\ & e_{i,j} + p_{j,k} \leq 1 + p_{i,k} \\ \forall k \in]i,j[& e_{k,j} + p_{i,k} \leq 1 + p_{i,j} \end{array} \right\} \text{Complex transitivity on } p \text{ and } e$$

$$\left\{ \begin{array}{lll} \forall k \in]j,n] & e_{i,j} + p_{i,j} + p_{j,k} \leq 1 + p_{i,k} + e_{i,k} \end{array} \right\} \text{Complex transivity on } s \text{ and } p$$

Lemma 1 (Completeness and correctness) *The set O contains one and only one point per distinct total preorder of n elements.*

The full proof (shown in Appendix A) proceeds by showing that the transitivity of the total preorder relation is preserved for all points in the set, i.e., all possible cases of transitivity have been enforced. Totality of the preorder comes from the mutual exclusion condition, and reflexivity is trivially satisfied. □

3.3 Pruning for Legality

The set O represents all possible and distinct fusion structures. We first prune it of all fusion structures that does not preserve the original program semantics. This to reduce the size of the search space, then containing only useful candidates. Such a pruning has a drastic impact: for instance, on the benchmark ludcmp, only 8 legal structures remain in place of the 10^{12} initial ones.

To the best of our knowledge, no previous work address the problem of constructing an enumerable search space of all and only legal fusion structures including interchange, skewing, shifting and peeling as enabling transformations for fusion. While the problem is combinatorial, we show how to control its complexity effectively.

3.3.1 The Algorithm

The general principle of the pruning algorithm is to detect all the smallest possible sets of unfusable statements $S_1, S_2, S_3, \ldots, S_n$, and for each of them, to update O by adding an affine constraint of the form:

$$e_{1,2} + e_{2,3} + \ldots + e_{n-1,n} < n-1 \tag{8}$$

thus preventing them (and any super-set of them) to be fused all together. We note $\mathcal{F}_1$ the final set with all pruning constraints for legality, $\mathcal{F}_1 \subseteq O$. A naive approach could be to enumerate all unordered subsets of the n statements of the program and check for fusability, while avoiding to enumerate a super-set of an unfusable set.

Instead, we propose to leverage the polyhedral dependence graph to test a much smaller set of possible structures. First, let us recall two intuitive properties on fusion. Given two statements R and S:

1. if R and S are not fusable, then any statement on which R transitively depends on is not fusable with S and any statement transitively depending on S;
2. reciprocally, if R and S must be fused, then any statement depending on R and on which S depends on must also be fused with R and S.

These properties allow to dramatically cut the number of tested sequences, in particular in highly constrained program such as in loop-intensive kernels. They are used at each step of the following algorithm.

The first step of our algorithm is to build a graph, called the *fusion graph* with one node per statement. We check the fusability of all possible pairs of statements, and add an edge between two nodes only if (1) there is a dependence between them, and (2) they can be fused leveraging Definition 1. Note that we do not need to check the fusability of non-dependent statements: they are trivially fusable under Definition 1.

Checking for fusability of two statements R and S is done by checking the existence of a solution in the set $\mathcal{T}_{R,S}$, defined as:

$$\mathcal{T}_{R,S} = \bigcap_{\forall \mathcal{D}_{R,S}} \left\{ \phi^S(\vec{x_S}) - \phi^R(\vec{x_R}) \geq \vec{0},\ \langle \vec{x_R}, \vec{x_S} \rangle \in \mathcal{D}_{R,S} \right\}$$

The second step is to enumerate all paths of length ≥ 2 in the fusion graph. Given a path p, the nodes in p represent a set of statements that has to be tested for fusability. Each time they are detected to not be fusable, all paths with p as a sub-path are discarded from enumeration, and $\mathcal{F}_1$ is updated with an equation in the form of (8).

Still, this algorithm implies to test for larger and larger set of statements, thus testing for the emptiness of increasingly larger $\mathcal{T}_{S1,\ldots,S_n}$ systems. To avoid performing this test, we want to define the transitivity of fusability, that is, given the fusability of each pairs of statements determine without solving $\mathcal{T}_{S1,\ldots,S_n}$ if they can be fused all together. One of the key problem is the loop permutation that can be required to fuse statements. As an illustration, consider the sequence of matrix-by-vector products $x = Ab,\ y = Ax,\ z = Ay$. While it is possible to fuse them 2-by-2, it is not possible to fuse them all together. When considering fusing loops for $x = Ab,\ y = Ax$, one has to permute loops in $y = Ax$. When considering fusing loops for $y = Ax,\ z = Ay$, one has to keep loops in $y = Ax$ as is.

To address this problem, we leverage the two following properties on fusion. First, Property 1 shows we can always find a complementary transformation to ϕ^s such that whatever the slowing, shifting or peeling applied to ϕ^R it can be compensated in ϕ^s to fuse R and S, if they are fusable at start.

Property 1 (Invariance of Fusability)
Given (ϕ^R, ϕ^S) *respecting (1). Then* $\forall\ \alpha_1 > 0,\ \beta_1$ *there exists* $\alpha_2 > 0,\ \beta_2$ *such that:*

$$(\phi^{R'}, \phi^{S'}) = (\alpha_1.\phi^R + \beta_1, \alpha_2.\phi^S + \beta_2)$$

also respects (1).

Then, Property 2 formalizes that, given a legal loop permutation that allows two statements to be fused, it is possible to fuse a third statement only if the same permutation has to be performed.

Property 2 (Transitivity of Fusability)
Given R, S and T three statements. If there exists valid hyperplanes (ϕ^R, ϕ^S), $(\phi^{S'}, \phi^T)$, $(\phi^{R'}, \phi^{T'})$ *such that:*

$$\begin{aligned}(\phi^R, \phi^S) &= (c_x^R.\vec{x}_R + \beta_1, c_y^S.\vec{x}_S + \beta_2)\\ (\phi^{S'}, \phi^T) &= (\alpha_1.c_y^S.\vec{x}_S + \beta_3, c_z^T.\vec{x}_T + \beta_4)\\ (\phi^{R'}, \phi^{T'}) &= (\alpha_2.c_x^R.\vec{x}_R + \beta_5, \alpha_3.c_z^T.\vec{x}_T + \beta_6)\end{aligned}$$

with $\alpha_i > 0,\ \beta_i \geq 0$. *Then R, S and T are fusable.*

Property 1 and Property 2 combined define the transitivity of fusability together with loop permutation (interchange), shifting and peeling as enabling transformations for fusion.

Leveraging Property 2, we label edges in the fusion graph with the list of all legal permutations (c_x^R, c_y^S) leading to fuse R and S. To check all legal (c_x^R, c_y^S), we simply test in $\mathcal{T}_{R,S}$ the existence of solutions of the form $(\alpha_1.c_x^R.\vec{x}_R + \beta_1,\ \alpha_2.c_y^S.\vec{x}_S + \beta_2)$, for all $x, y \geq 1$. When a skew is necessary to fuse R and S, the edge is labeled with *skew* instead. Then, to detect if a group of statements is fusable, we only have to check for the existence of a compatible permutation along all the edges in p, avoiding to build and test in $\mathcal{T}_{S_1 \ldots S_n}$. Note that if a *skew* edge is in the path, we still need to perform the check on the full system, as Property 2 does not determine fusability along with skewing.

In practice this algorithm proved to be very efficient, for the ludcmp benchmark the space is pruned from 10^{12} candidates to 8 remaining legal ones in less than $0.4s$ on an Intel Xeon $2.4\,GHz$.

3.3.2 Generalization to Multi-Level Fusion

The proposed algorithm builds the set $\mathcal{F}_1$ of legal fusion structures at the first dimension, that is, resulting fusion structures represent partitions where statements in a class share at least one (but not necessary more) common loop. To build candidates fusion structures for more than one level, we proceed recursively by building $\mathcal{F}_k$ ($k = 1$ at start), picking a candidate in $\mathcal{F}_k$, updating the dependence graph by removing all dependences satisfied at loop level k, then build $\mathcal{F}_{k+1}$ from the updated dependence graph. Note that it is possible to compute a convex set $\mathcal{F}_{1,\ldots,n}$ containing all legal multi-level fusion structures . To do so, we reproduce the layout of O for the desired number of levels, and add inter-dimension consistency conditions of the form:

$$p^k_{i,j} = 1 \Rightarrow p^l_{i,j} = 1,\ \forall l > k$$

Then, we use the previous algorithm to constrain the space. It is redundant to do so, as it also implies to enumerate all legal fusion structures at a given depth to be able to constrain the next depth. We therefore use the recursive procedure depicted above for the enumeration.

Studies performed on the performance impact of selecting schedules at various levels highlighted the much higher impact of carefully selecting outer loops [23, 24]. Hence, the selection of the fusion structure at the outermost level captures the most significant difference in terms of locality and communication. It is thus possible to limit the recursive traversal of fusion structures to the outer loop level only while still obtaining significant performance improvement and a wide range of transformed codes. Nevertheless, when the number of candidates in $\mathcal{F}_1$ is very small typically because of several loop-dependent dependences at the outer level, it is relevant to build $\mathcal{F}_2$ and further. One can choose to enumerate the next dimension if there are 2 or less candidates at the current dimension, mostly to offer freedom for the iterative search while still controlling the combinatorial nature of the recursive search. Note that in the experiments presented in this paper we only traverse $\mathcal{F}_1$.

3.3.3 Search space statistics

Several $e_{i,j}$ and $p_{i,j}$ variables are set during the pruning of O, so several consistency constraints are made useless and are not built, significantly helping to reduce the size of the space to build. Table 3 illustrates this by highlighting, for our benchmarks considered, the properties of the polytope O in terms of the number of dimensions (#dim), constraints (#cst) and points (#points) when compared to $\mathcal{F}_1$, the polytope of legal fusion structures at the first dimension. We also report some information on the considered SCoP for each benchmark (#loops the number of loops, #stmts the number of statements, #refs the number of array references).

3.4 Testing Relevant Program Versions

A consequence of fusion is to increase cache load, that may affect performance. There is usually a lack of performance improvement when two statements that do not reference any common memory location are fused, and with the risk of observing a performance degradation if cache conflicts occur. To avoid testing such cases, we perform a second level of pruning and remove all candidate fusion structures such that statements with no reuse are fused. This again to allow the iterative search to quickly focus on relevant candidates only. Note that maximal fusion may thus be removed from the search space.

				$\mathcal{O}$			$\mathcal{F}_1$		
Benchmark	#loops	#stmts	#refs	#dim	#cst	#points	#dim	#cst	#points
advect3d	12	4	32	12	58	75	9	43	26
atax	4	4	10	12	58	75	6	25	16
bicg	3	4	10	12	58	75	10	52	26
gemver	7	4	19	12	58	75	6	28	8
ludcmp	9	14	35	182	3003	$\approx 10^{12}$	40	443	8
doitgen	5	3	7	6	22	13	3	10	4
varcovar	7	7	26	42	350	47293	22	193	96
correl	5	6	12	30	215	4683	21	162	176

Figure 3: Properties of the Search Space

Finally, for each remaining candidate fusion structure, we also test with and without the application of polyhedral tiling. The motivation is twofold. Firstly, tiling may be detrimental as it may introduce complex loop bounds and the computation overhead may not be compensated by the locality improvement. Secondly, tiling may prevent the compiler from performing aggressive, low-level optimizations, as current production compilers optimization heuristics are still very conservative, in particular when loop bounds are complex as in polyhedrally tiled code.

In Figure 4 The final number of candidates that end up being tested during the iterative process is reported (#Tested), as well as the time to build all candidate fusion structures from the input source code (that is, including all analysis) on an Intel Xeon $2.4\,GHz$. We also report the dataset size we used for the benchmarks (Pb. Size).

Benchmark	#loops	#stmts	#refs	#Tested	Time	Pb. Size
advect3d	12	4	32	52	0.82s	300x300x300
atax	4	4	10	20	0.06s	8000x8000
bicg	3	4	10	32	0.05s	8000x8000
gemver	7	4	19	12	0.06s	8000x8000
ludcmp	9	14	35	2	0.54s	1000x1000
doitgen	5	3	7	8	0.08s	50x50x50
varcovar	7	7	26	88	0.09s	1000x1000
correl	5	6	12	104	0.09s	1000x1000

Figure 4: Search space statistics

4 Experimental Results

The automatic optimization and parallelization process has been implemented in PoCC, the *Polyhedral Compiler Collection*, a complete source-to-source polyhedral compiler based on well established Free software for polyhedral compilation.[2] Specifically, the search space construction has been implemented in the LetSee optimizer and the transformations for tiling and parallelization are computed by the Pluto optimizer. In the generated programs, parallelization is obtained by marking transformed loops with OpenMP pragmas. In addition, when compiling for SIMD-capable targets, intra-

[2] A Beta version of PoCC is available on `http://pocc.sourceforge.net`

tile parallel loops are moved to the innermost position and when compiling with ICC marked with `ivdep` pragma to facilitate compiler auto-vectorization, when possible.

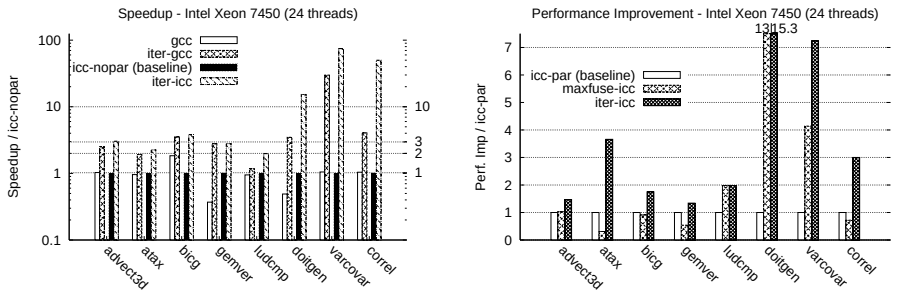
4.1 Experimental Setup

We experimented on three high-end machines: a 4-socket Intel hexa-core Xeon E7450 (Dunnington) at 2.4 *GHz* with 64 *GB* of memory (24 cores, 24 hardware threads), a 4-socket AMD quad-core Opteron 8380 (Shanghai) at 2.50 *GHz* (16 cores, 16 hardware threads) with 64 *GB* of memory, and an 2-socket IBM dual-core Power5+ at 1.65 *GHz* (4 cores, 8 hardware threads) with 16 *GB* of memory.

All systems were running Linux 2.6.x. We used Intel ICC 11.0 with options `-fast -parallel -openmp` referred to as icc-par, and with `-fast` referred to as icc-nopar, GCC 4.3.3 with options `-O3 -msse3 -fopenmp` as gcc, and IBM/XLC 10.1 compiled for Power5 with options `-O3 -qhot=nosimd -qsmp -qthreaded` referred to as xlc-par, and `-O3 -qhot=nosimd` referred as xlc-nopar.

We consider 8 benchmarks, typical from compute-intensive sequences of algebra operations. atax, bicg and gemver are compositions of BLAS operations [22], ludcmp solves simultaneous linear equations by LU decomposition, advect3d is an advection kernel for weather modeling and doitgen is an in-place 3D-2D matrix product. correl creates a correlation matrix, and varcovar creates a variance-covariance matrix, both are used in Principal Component Analysis in the StatLib library. Problem sizes are reported in column Pb. Size of Figure 4.

The time to compute the space, pick a candidate and compute a full transformation is negligible with respect to the compilation and execution time of the tested versions. In our experiments, the full iterative compilation process takes a few seconds for the smaller benchmarks, and up to about 1 minute for correl on Xeon.



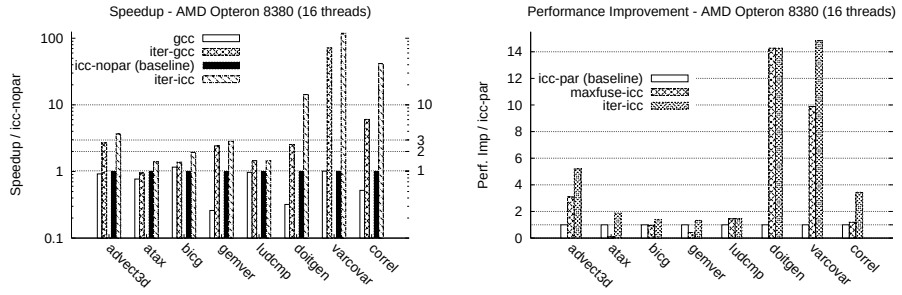
(a) Speedup over best single-threaded version

(b) Performance improvement over maximal fusion and over the reference auto-parallelizing compiler

Figure 5: Performance Results for the quad-Intel E7450

4.2 Performance Improvement

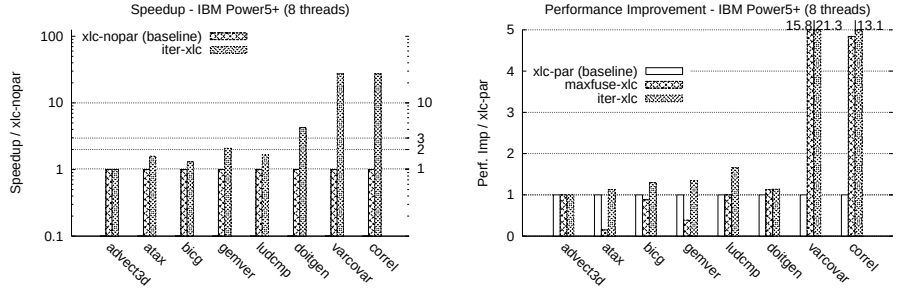
In Figures 5, 6 and 7, we report for all benchmarks the speedup of our iterative technique when used on top of the three compilers (iter-icc, iter-gcc and iter-xlc) normalized



(a) Speedup over best single-threaded version

(b) Performance improvement over maximal fusion and over the reference auto-parallelizing compiler

Figure 6: Performance Results for the quad-AMD Opteron 8380



(a) Speedup over best single-threaded version

(b) Performance improvement over maximal fusion and over the reference auto-parallelizing compiler

Figure 7: Performance Results for the bi-IBM Power5+

to the best single-threaded version produced by the native compiler (ICC for Intel and Opteron, XLC for Power5+). We also compare the performance improvement obtained over maximal fusion [6] and over ICC/XLC with automatic parallelization (icc-par or xlc-par) in those figures.

For doitgen, correl and varcovar, three compute-bound benchmarks, our technique exposes a program with a significant parallel speedup of up to 112× on Opteron. Our optimization technique goes far beyond parallelizing programs, and for these benchmarks locality and vectorization improvements were achieved by our framework. For advect3d, atax, bicg, and gemver we also observe a significant speedup, but this is limited by memory bandwidth as these benchmarks are memory-bound. Yet, we are able to achieve a solid performance improvement for those benchmarks over the native compilers, of up to 3.8× for atax on Xeon and 5× for advect3d on Opteron. For ludcmp, although parallelism was exposed, the speedup remains limited as the program

offers little opportunity for high-level optimizations. Yet, our technique outperforms the native compiler, by a factor up to 2× on Xeon.

For Xeon and Opteron, the iterative process outperforms ICC with auto parallelization, with a factor between 1.2× for gemver on Intel to 15.3× for doitgen. For both of these kernels, we also compared with an implementation using Intel Math Kernel Library (MKL) 10.0 and AMD Core Math Library (ACML) 4.1.0 for the Xeon and Opteron machines respectively, and we obtain a speedup of 1.5× to 3× over these vendor libraries. For varcovar, our technique outperforms the native compiler by a factor up to 15×. Although maximal fusion significantly improved performance, the best iteratively found fusion structure allows for a much better improvement, up to 1.75× better. Maximal fusion is also outperformed for all but ludcmp. This highlights the power of the method to discover the right balance between parallelism (both coarse-grain and fine-grain) and locality.

On Power5+, on all but advect3d the iterative process outperforms XLC with auto-parallelization, by a factor between 1.1× for atax to 21× for varcovar.

For the sake of completeness, we also provide the best performance in GFLOP/s for all our benchmarks in Figure 8. Benchmarks are superscripted with ^{d} when the data type is *double*, and with ^{f} for *float*.

Benchmark	Xeon E7450 (24 cores)	Opteron 8380 (16 cores)	Power5+ (4 cores)
advect3d^f	0.47	0.53	0.34
atax^d	2.13	1.42	1.16
bicg^d	2.13	1.70	1.15
gemver^d	2.20	2.66	1.42
ludcmp^d	1.33	0.75	0.48
doitgen^d	44.64	31.25	10.41
correl^f	16.71	11.14	7.16
varcovar^f	50.05	33.36	14.30

Figure 8: Best Performance obtained, in GFLOP/s

4.3 Performance Portability

Beyond absolute performance improvement, another motivating factor for iterative selection of fusion structures is performance portability. Because of significant differences in design, in particular in SIMD units' performance and cache behavior, a transformation has to be tuned for a specific machine. This leads to a significant variation in performance across tested frameworks.

To illustrate this, we show in Figure 9 the relative performance normalized with respect to icc-par of gemver, for Intel and Opteron. The version index is plotted on the x axis, 1 is max-fuse and 8 is maximal distribution.

For Xeon, the best version is 4, corresponding to the fusion structure in Figure 2. It performs 10% better than version 2 — version 2 corresponds to the fusion structure $(0,0,0,1)$ — which is the optimal fusion for Opteron. And for the Opteron, version 4 performs 20% slower than 2. Note that on gemver the performance distribution for Power5+ is very similar as for Xeon.

Performance variation is also exhibited by maximal fusion results over the three architectures. For ludcmp, while it is the best performing one on Xeon and Opteron, it is not on Power5+. Such a pattern can also be observed for doitgen.

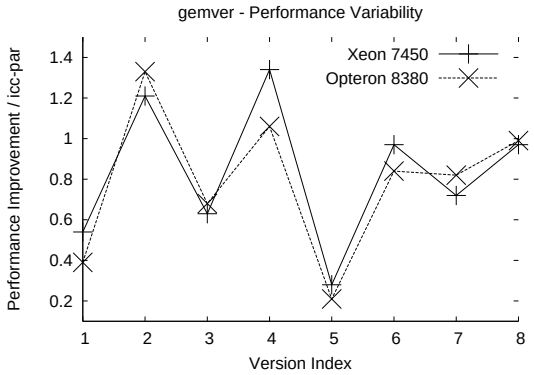


Figure 9: Performance variability for gemver

The trade-off between coarse-grain parallelization and vectorization is very difficult to capture, as it also depends on the capability of the back-end compiler to perform vectorization. One has to capture the interplay between distinct optimization passes, something missing in present day compilers. Moreover, accurate profitability models have to be relied upon, and their design remains a major challenge for compiler designers. Tuning the trade-off between fusion and distribution is a relevant technique to address the performance portability issue. Our technique is able to automatically adapt to the target framework, and successfully discovers the optimal fusion structure, whatever the specifics of the program, compiler and architecture.

Finally, note that by checking against the best performing version found on the entire $\mathcal{F}_1$ set, we have experimentally validated that the pruning strategy discussed in Section 3.4 does not prevent us from finding the optimal fusion structure for all tested benchmarks and architectures.

5 Related Work

Traditional works on loop fusion [18, 20, 21, 28] are restricted in their ability to find complex fusion structures. This is mainly due to the lack of a powerful representation for dependences and transformations. Hence, non-polyhedral approaches typically study fusion in isolation from other transformations. Megiddo and Sarkar [21] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed. Decoupling parallelization and fusion clearly misses several interesting solutions that would have been captured if the legal fusion choices were itself cast into their framework. Darte et al. [10, 9] studied fusion for data-parallelization, but only in combination with shifting. In contrast to all of these works, our search space can enable fusion in the presence of all polyhedral transformations.

Several heuristics for loop fusion and tiling have been proposed [33, 26]. Yet those heuristics fail to capture the heavy interplay between loop transformations, back-end optimizations performed by the compiler, and components of the target architecture.

The polyhedral model creates many more opportunities for the construction of loop nest optimizers and parallelizing compilers. It is currently being integrated in production compilers, including GCC[3] and IBM XL.

Bondhugula et al. designed Pluto, the first integrated fusion and tiling heuristic based on the polyhedral model [5, 6], and subsuming a large space of additional loop transformations (interchange, skewing, shifting). It inherits the flexibility of the tiling hyperplane method [16, 15] to build complex sequences of enabling and communication-minimizing transformations. Despite the weaknesses of its target-independent optimization model, it does identify interesting parallelism-locality trade-offs.

Powerful semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [17, 8, 14, 7, 31]. They capture fusion structures, but neither do they define automatic iteration schemes nor do they integrate a model-based heuristic to construct profitable parallelization and tiling strategies.

Iterative compilation has proved its efficiency in providing solid performance improvements over a broad range of architectures and transformations [4, 30, 1, 19, 26, 13, 24, 32]. However, none of the previous works achieved the expressiveness and application of complex transformation sequences presented in this paper, along with a focused search on legal candidates only.

6 Conclusion

This paper addressed the problem of optimizing and parallelizing programs automatically, focusing on static control loop nests. Our approach departs from the traditional best-effort compiler optimizations, aiming for effective portability of performance over a variety of shared-memory multiprocessors. We proposed a hybrid iterative and model-driven approach, leveraging a state-of-the-art parallelization method based on loop tiling, and combining it with a feedback-directed scheme for loop fusion and distribution. Our technique builds an expressive search space of loop transformation sequences, expressed in the polyhedral model as a set of affine scheduling functions. The search space encompasses complex compositions of loop transformations, including loop fusion and distribution, loop tiling for parallelism and locality (caches, registers), loop interchange, and loop shifting (pipelining). We proposed a convex encoding of all legal transformed program versions as the space to search. We performed experiments on three different platforms: a new 24-core Xeon and a 16-core Opteron, and a 4-core Power5+. Our experiments confirm that a single program version does not perform equally well on different targets, with penalties reaching $2\times$ when running the best version for a given target on a different target. We also consistently demonstrate strong scalability on kernels where state-of-the-art model-based compilers could not find significant coarse-grain parallelism, with performance improvement factors up to $15.3\times$ over Intel's compiler and up to $21.3\times$ over IBM's compiler. In the future, we will study the applicability of machine learning techniques to prune our hybrid optimization space or predict the performance of transformed program versions. We will also continue to look for ways of building an even more expressive space, and narrowing down the gap with respect to peak performance on a wide set of benchmarks and target architectures.

[3] Graphite development branch: `http://gcc.gnu.org/wiki/Graphite`.

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, France, Sept. 2004.

[3] M.-W. Benabderrahmane, C. Bastoul, L.-N. Pouchet, and A. Cohen. A conservative approach to handle full functions in the polyhedral model. Technical Report 6814, INRIA Research Report, January 2009. Submitted for publication.

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

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, Apr. 2008.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[7] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.

[8] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.

[9] A. Darte and G. Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.

[10] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Proc. Letters*, 7(4):379–392, 1997.

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

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

[13] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation (PLDI'05)*, pages 315–326. ACM, 2005.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.

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

[16] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.

[17] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.

[18] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, 1993.

[19] S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *Proc. of the 2005 Intl. Conf. on Parallel Processing Workshops (ICPPW'05)*, pages 137–144, Washington, DC, USA, 2005. IEEE Comp. Soc.

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

[21] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

[22] B. Norris, A. Hartono, E. Jessup, and J. Siek. Generating empirically optimized composed matrix kernels from MATLAB prototypes. In *Int. Conf. on Computational Science (ICCS'09)*, may 2009.

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

[24] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'08)*, pages 90–100. ACM Press, 2008.

[25] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proc. of the IEEE/ACM Fifth Intl. Symp. on Code Generation and Optimization (CGO'07)*, pages 144–156. IEEE Comp. Soc. press, 2007.

[26] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proc. of the 20th Intl. Conf. on Supercomputing (ICS'06)*, pages 249–258. ACM press, 2006.

[27] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[28] S. Singhai and K. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[29] N. J. A. Sloane. Sequence a000670. The On-Line Encyclopedia of Integer Sequences, `http://www.research.att.com/~njas/sequences/A000670`.

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

[31] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.

[32] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *Intl. Symp. on Code Generation and Optimization (CGO'09)*, Mar. 2009.

[33] M. Wolf, D. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.

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

A Proof of Lemma 1

We restate the expression of O, the convex set of all, distinct total preorders of n elements. For $1 \leq i < n,\ i < j \leq n,\ O$ is:

$$\left\{\begin{array}{lll}
 & 0 \leq p_{i,j} \leq 1 \\ & 0 \leq e_{i,j} \leq 1 \end{array}\right\} \text{Variables are binary}$$

$$\left\{\begin{array}{lll}
 & p_{i,j} + e_{i,j} \leq 1 \end{array}\right\} \text{Relaxed mutual exclusion}$$

$$\left\{\begin{array}{lll}
\forall k \in]j,n] & e_{i,j} + e_{i,k} \leq 1 + e_{j,k} \\ & e_{i,j} + e_{j,k} \leq 1 + e_{i,k} \end{array}\right\} \text{Basic transitivity on } e$$

$$\left\{\begin{array}{lll}
\forall k \in]i,j[& p_{i,k} + p_{k,j} \leq 1 + p_{i,j} \end{array}\right\} \text{Basic transitivity on } p$$

$$\left\{\begin{array}{lll}
\forall k \in]j,n] & e_{i,j} + p_{i,k} \leq 1 + p_{j,k} \\ & e_{i,j} + p_{j,k} \leq 1 + p_{i,k} \\ \forall k \in]i,j[& e_{k,j} + p_{i,k} \leq 1 + p_{i,j} \end{array}\right\} \text{Complex transitivity on } p \text{ and } e$$

$$\left\{\begin{array}{lll}
\forall k \in]j,n] & e_{i,j} + p_{i,j} + p_{j,k} \leq 1 + p_{i,k} + e_{i,k} \end{array}\right\} \text{Complex transivity on } s \text{ and } p$$

We want to prove that *the set O contains one and only one point per distinct total preorder of n elements.*

Proof:

We first prove that O contains all and only total preorders, before proving the uniqueness of preorders in O.

From the encoding through s, p and e variables chosen, at least all total preorders are represented in the initial set

$$\left\{\begin{array}{l} 0 \leq p_{i,j} \leq 1 \\ 0 \leq e_{i,j} \leq 1 \\ 0 \leq s_{i,j} \leq 1 \end{array}\right\}$$

We now show that the successive constraints added to prune the set remove all points that are not a valid total preorder. To prove so, we rely on the fact that a total preorder relation is a relation which is total, transitive and symetric. Hence, we prove that our constraints are sufficient to guarantee to preserve the totality, the transitivity and the reflexivity of the relation.

Totality: Given $x,\ y$ two elements of a set S of n elements on which the total preorder relation is defined. Without any loss of generality and for the rest of the proof, we assume that S is the set of consecutive integers from 1 to n. Given $a,\ b$ their position identifier as specified by the preorder. Totality gives:

$$a \preceq b \ \vee \ b \preceq a$$

Either $a < b$, $a = b$, $b < a$ or $b = a$ which is equivalent to $a = b$. This is guaranteed by the relaxed mutual exclusion inequality.

Transitivity: Given $x,\ y,\ z$ three elements and $a,\ b,\ c$ their respective partition identifier. Transitivity gives:

$$a \preceq b \ \wedge\ b \preceq c \ \Rightarrow\ a \preceq c$$

That is, one of the following configuration must occur:

1. $a < b \ \wedge\ b < c \ \Rightarrow\ a < c$
2. $a < b \ \wedge\ b = c \ \Rightarrow\ a < c$
3. $a = b \ \wedge\ b < c \ \Rightarrow\ a < c$
4. $a = b \ \wedge\ c < b \ \Rightarrow\ c < a$
5. $a = b \ \wedge\ b = c \ \Rightarrow\ a = c$
6. $b < a \ \wedge\ c < b \ \Rightarrow\ c < a$
7. $b < a \ \wedge\ b = c \ \Rightarrow\ c < a$

Converting 1. into our encoding gives:

$$p_{x,y} \ \wedge\ p_{y,z} \ \Rightarrow p_{x,z} \tag{9}$$

To generalize this constraint to the n possible elements, we must then consider the different possible values for $x,\ y,\ z$: we can have $x < y$ or $x > y$, $x < z$ or $x > z$, and $y < z$ or $z < y$. We start by focusing only on the case where $x < y < z$. (9) is written:

$$\forall\ 1 \leq i < k < j \leq n, \quad p_{i,k} \ \wedge\ p_{k,j} \ \Rightarrow p_{i,j} \tag{10}$$

This equation can be converted in an affine form in a deterministic fashion (one can use a Boolean table to ensure the constraint defines an equivalent logic as the implication), and once converted in an affine form corresponds to the basic transitivity of p coefficients inequalities shown in the definition of O.

Converting 5. into our encoding gives:

$$e_{x,y} \ \wedge\ e_{y,z} \ \Rightarrow e_{x,z} \tag{11}$$

For this case, if $x < y < z$, then (11) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \ \wedge\ e_{j,k} \ \Rightarrow e_{i,k}, \tag{12}$$

If $y < x < z$, then (11) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \ \wedge\ e_{i,k} \ \Rightarrow e_{j,k}, \tag{13}$$

These equations once converted in an affine form correspond to the basic transitivity of e coefficients.

Converting 3. into our encoding gives:

$$e_{x,y} \ \wedge\ p_{y,z} \ \Rightarrow p_{x,z} \tag{14}$$

If $x < y < z$, then (14) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \ \wedge\ p_{i,k} \ \Rightarrow p_{j,k}, \tag{15}$$

If $y < x < z$, then (14) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \wedge p_{j,k} \Rightarrow p_{i,k}, \tag{16}$$

Converting 2. into our encoding gives:

$$e_{y,z} \wedge p_{x,y} \Rightarrow p_{x,z} \tag{17}$$

If $x < z < y$, then (17) is written:

$$\forall 1 \leq i < k < j \leq n, \quad e_{k,j} \wedge p_{i,k} \Rightarrow p_{i,j}, \tag{18}$$

Equations (15), (16) and (18) correspond to the complex transitivity constraints on the p and e variables.

Converting 6. into our encoding gives:

$$s_{x,z} \wedge p_{y,z} \Rightarrow s_{x,y} \tag{19}$$

If $x < y < z$, then (19) is written (thanks to the substitution coming from the mutual exclusion equation):

$$\forall 1 \leq i < j < k \leq n, \quad \neg e_{i,j} \wedge \neg p_{i,j} \wedge p_{j,k} \Rightarrow \neg p_{i,k} \wedge \neg e_{i,k}, \tag{20}$$

This equations corresponds to the complex transitivity constraints on the p and s variables.

Several cases have not been explicitely addressed, because they are equivalent to the above-mentionned affine constraints. Their enumeration and computing their equivalence with the presented cases is left to the motivated reader.

All necessary conditions for transitivity have been enforced in O.

Reflexivity: Reflexivity is trivially satisfied in our encoding.

This concludes proving that all points in O is a total preorder, and that all total preorders are in O.

We must now prove that there is only one point in O per distinct total preorder.

Uniqueness: To prove so, we show there exists a bijection $f : \mathcal{P} \rightarrow O$ between the set of distinct total preorders $\mathcal{P}$ and O.

We first prove that it is not possible that two distinct preorders are represented by the same point in O. Suppose there exists two distinct total preorders p_1 and p_2 such that

$$f(p_1) = f(p_2) \wedge p_1 \neq p_2$$

By construction of the encoding, two distinct preorders result in at least one modification of a variable ($e_{i,j}$ and/or $p_{i,j}$) used to encode the preorder. Hence we must have:

$$f(p_1) = f(p_2) \Rightarrow p_1 = p_2$$

To show that it is not possible to have two distinct points in O representing the same total preorder, we again rely on our encoding definition.

This concludes the proof of Lemma 1. □

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399