

# Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework

Louis-Noël Pouchet<sup>1,3</sup>  
Cédric Bastoul<sup>3</sup> Albert Cohen<sup>3</sup>

Uday Bondhugula<sup>2</sup>  
J. Ramanujam<sup>4</sup> P. Sadayappan<sup>1</sup>

<sup>1</sup> The Ohio State University

<sup>2</sup> IBM T.J. Watson Research Center

<sup>3</sup> INRIA and Paris-Sud 11 University

<sup>4</sup> Louisiana State University

**Abstract**—Today’s multi-core era places significant demands on an optimizing compiler, which must (i) parallelize programs, (ii) exploit the memory hierarchy, and (iii) leverage the ever-increasing SIMD capabilities of modern processors. Existing model-based heuristics used in compilers are unable to identify profitable parallelism/locality trade-offs, and usually lead to sub-optimal performance.

To address this problem, we distinguish optimizations for which effective model-based heuristics and profitability estimates exist, from those optimizations that require empirical search to achieve good performance in a portable fashion. Using this, we have developed a complete automatic framework in which we focus the empirical search on the set of valid possibilities to perform fusion/code motion, resorting to model-based mechanisms for the task of performing tiling, vectorization or parallelization on the transformed program. We demonstrate the effectiveness of this approach both in terms of strong performance improvements and adaptability to the target architecture.

## I. INTRODUCTION

Portability of performance over a broad range of architectures is a very difficult task. It requires: (1) to be able to apply complex sequences of loop transformations; and (2) to precisely 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 to express 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 [16], [26]. It is also hard 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 [7], we show that a purely analytical approach fails to achieve portability of performance. It is not sufficient since several high-impact factors, including cache conflicts, memory bandwidth 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 for any shared memory target.

To address these challenges, we designed a combined iterative and model-driven scheme for optimization and parallelization. It relies on an iterative, feedback-directed exploration of loop fusion/distribution choices. In turn, each fusion/distribution choice drives model-based algorithms for many other loop transformations, including loop tiling and vectorization. Portability of performance is achieved thanks to iteratively testing different program versions: our method finds the optimal version on all benchmarks we considered. We obtained improvements ranging from  $1\times$  to  $8.5\times$  over reference parallelizing compilers using model-based heuristics, and validated the portability of our approach considering two modern multi-core architectures, Intel Dunnington and AMD Shanghai, and a low-power embedded Intel Atom processor. The memory hierarchy and interconnect of these multiprocessor architectures are completely different (front-side bus vs. point-to-point links); they were chosen to emphasize performance portability issues and stress the sensitivity of our method to the memory hierarchy.

The rest of the paper is organized as follows. Section II details the motivations and problem statement. Section III recalls the fundamental concepts in polyhedral compilation. Section IV details the search space construction, pruning and traversal strategy. Section V presents the model-based optimization algorithms used in our hybrid strategy. Section VI evaluates this technique experimentally. Section VII discusses related work, before the conclusion in Section VIII.

## II. PROBLEM STATEMENT

The efficient mapping of a computation kernel to a modern multi-core architecture requires the synergistic operation of many hardware components in the chip. This typically covers the exploitation of

- thread-level parallelism;
- the memory hierarchy, including prefetch units, different cache levels, memory buses and interconnect; (and)
- all available computational units, including SIMD units.

Because of the very complex interplay between all these components, translating specific properties on the input code, such as the number of parallel loop iterations for instance, into actual performance metrics is a severe challenge. Considering several high-level program transformations which expose the same amount of thread-level parallelism, it is currently out

of reach to *guarantee at compile-time* which of these leads to the maximal performance. The explanation resides in the combined effectiveness of other components, such as the memory hierarchy and the vector units, which can be significantly different from one version to another.

To maximize performance one must carefully tune the trade-off between the different levels of parallelism and the usage of the local memory components. Maximizing the locality of data may be detrimental to inner-parallelism, or at least contradictory with an efficient, well-aligned inner-loop vectorization. On the other hand, driving the optimization towards the most efficient vectorization may require excessive loop distribution, resulting in a poor memory reuse and thus may jeopardize performance.

We characterize the main program transformations a loop nest optimizer should combine and carefully balance as follows:

- thread-level parallelism extraction, to expose coarse-grain parallelism and benefit from the different hardware threads available on the chip(s);
- loop tiling, to improve the locality of computation and reduce the number of cache misses;
- SIMD-level parallelism extraction, to expose inner loops which can produce efficiently vectorized loops;

Most previous work take a multi-stage, decoupled approach to select optimization: they first identify a transformation to expose thread-level parallelism, then try to apply tiling on the resulting code, and finally try to vectorize the final output [20], [2]. Recent work by Bondhugula et al. [7] integrated the extraction of parallelism and the exposition of tilable loop nests, but it did not take the memory hierarchy into account when selecting which loops to fuse and to parallelize, and it did not consider the effect of these transformations on SIMD parallelism. As a result of this approach, reuse is maximized and parallel tiled code is generated, but this can lead to sub-optimal performance since maximizing locality can prohibit the vectorization of inner-loops and increase cache interference conflicts.

It leads to the key observation that *loop fusion and distribution drive the success of subsequent optimizations, such as vectorization, tiling or array contraction*: the number of vectorizable or tilable loops is related to the set of statements which are fused under a common loop. The loop structure can be seen as resulting from a *partitioning of the program*, where the statements in the same class of the partition all share at least one common outer-loop. To illustrate this, we propose to study the performance of different ways to partition the program, emphasizing the need for a target-specific tuning of the program partition. Let us consider the example of a sequence of 2 matrix multiplications,  $D = A.B.C$ , as shown in Figure 1.

We present in Figure 2 the result of a purely model-driven approach geared towards (1) minimizing communication and maximizing the data locality for the full program, and (2) exposing thread-parallelism and tilable loops [7]. This corresponds to partitioning the program such that all statements are in the same class:  $p_{maxfuse} = \{\{R, S, T, U\}\}$ . A complex sequence of loop transformations that includes skewing is

```

for (i1 = 0; i1 < N; ++i1)
  for (j1 = 0; j1 < N; ++j1) {
R   tmp[i1][j1] = 0;
    for (k1 = 0; k1 < N; ++k1)
S     tmp[i1][j1] += A[i1][k1] * B[k1][j1];
  }
  for (i2 = 0; i2 < N; ++i2)
    for (j2 = 0; j2 < N; ++j2) {
T     D[i2][j2] = 0;
      for (k2 = 0; k2 < N; ++k2)
U       D[i2][j2] += tmp[i2][k2] * C[k2][j2];
    }
  }

```

Fig. 1. Original code:  $tmp = A.B$ ,  $D = C.tmp$

needed to implement this partitioning. Note that for this example and the following, we omit the insertion of pragmas for OpenMP parallelization and vectorization as well as the tiling of the outer-most loop(s), for the sake of readability.

```

parfor (c0 = 0; c0 < N; c0++) {
  for (c1 = 0; c1 < N; c1++) {
R   tmp[c0][c1]=0;
T   D[c0][c1]=0;
    for (c6 = 0; c6 < N; c6++)
S     tmp[c0][c1] += A[c0][c6] * B[c6][c1];
    parfor (c6 = 0; c6 <= c1; c6++)
U     D[c0][c6] += tmp[c0][c1-c6] * C[c1-c6][c6];
  }
  for (c1 = N; c1 < 2*N - 1; c1++)
    parfor (c6 = c1-N+1; c6 < N; c6++)
U     D[c0][c6] += tmp[c0][c1-c6] * C[c1-c6][c6];
}

```

Fig. 2. Minimal communication, maximal fusion ( $p_{maxfuse}$ )

Considering a 4-socket Intel Xeon hexa-core 7450 server (24 cores, Dunnington microarchitecture), this transformation leads to a  $2.4\times$  speedup over Intel’s compiler ICC 11.1 with automatic parallelization enabled, with  $N=1024$  using double-precision arithmetic.

We now observe that there exist 12 different valid partitionings for this program, i.e., all possible ways to combine  $R$ ,  $S$ ,  $T$ ,  $U$  provided  $R$  is before  $T$ ,  $T$  is before  $U$ , and  $S$  is before  $U$ .  $p_{xeon} = \{\{R\}, \{S\}, \{T\}, \{U\}\}$  is one, and is shown in Figure 3. This partitioning no longer minimizes communication/synchronization, but on the other hand allows us to expose inner parallel loops for all statements. This leads to a  $3.9\times$  speedup over the original code, and is the best partitioning for this machine.

```

parfor (i1 = 0; i1 < N; ++i1)
  parfor (j1 = 0; j1 < N; ++j1)
R   tmp[i1][j1] = 0;
  parfor (i1 = 0; i1 < N; ++i1)
    for (k1 = 0; k1 < N; ++k1)
      parfor (j1 = 0; j1 < N; ++j1)
S        tmp[i1][j1] += A[i1][k1] * B[k1][j1];
  parfor (i2 = 0; i2 < N; ++i2)
    parfor (j2 = 0; j2 < N; ++j2)
T      D[i2][j2] = 0;
  parfor (i2 = 0; i2 < N; ++i2)
    for (k2 = 0; k2 < N; ++k2)
      parfor (j2 = 0; j2 < N; ++j2)
U        D[i2][j2] += tmp[i2][k2] * C[k2][j2];

```

Fig. 3. Best loop structure for Intel Xeon 7500 with ICC ( $p_{xeon}$ )

Considering now a 4-socket AMD Opteron quad-core 8380 server (16 cores, Shanghai microarchitecture), the best par-

tioning is  $p_{opteron} = \{\{R\}, \{T, S\}, \{U\}\}$ , and is shown in Figure 4.

```

parfor (c1 = 0; c1 < N; c1++)
  parfor (c2 = 0; c2 < N; c2++)
R   C[c1][c2] = 0;
  parfor (c1 = 0; c1 < N; c1++)
    for (c3 = 0; c3 < N; c3++) {
T     E[c1][c3] = 0;
      parfor (c2 = 0; c2 < N; c2++)
S     C[c1][c2] += A[c1][c3] * B[c3][c2];
    }
  parfor (c1 = 0; c1 < N; c1++)
    for (c3 = 0; c3 < N; c3++)
      parfor (c2 = 0; c2 < N; c2++)
U     E[c1][c2] += C[c1][c3] * D[c3][c2];

```

Fig. 4. Best loop structure for AMD Opteron with ICC ( $p_{opteron}$ )

This partitioning, when used on the Intel Xeon, performs 20% slower than the partitioning in Figure 3. To further emphasize the impact of the partitioning on performance, we performed a similar analysis on a low-power Intel Atom 230 processor (single-core, 2 hardware threads, Diamondville microarchitecture). For this case, the best found partitioning is  $p_{atom} = \{\{R, T\}, \{S, U\}\}$ , for a 10% improvement over  $p_{opteron}$  and  $p_{xeon}$ , and a  $3.5\times$  improvement over  $p_{maxfuse}$ . Data locality and vectorization dominates on the Atom, leading to a completely different optimal partitioning.

We summarize these results in Figure 5 where for these three architectures, we report the performance improvement over the original code (Improv.), and the performance improvement over the partitioning which performs on average the best on the three machines (Variability).

	Xeon	Opteron	Atom
Improv.	3.6×	8.3×	31.3×
Variability	20%	11%	14%

Fig. 5. Performance improvement and variability

The main performance difference between these program versions is not the exploitation of a single hardware component, but how parallelization, vectorization and data cache utilization interacted between each other; in other words how efficient was the synergistic usage of hardware resources. However what drives this interaction from the point of view of high-level program transformations is the loop nest organization resulting from the program partitioning.

Our approach for program optimization decouples the search of the *program structure* from the application of performance-enabling transformations, such as locality improvement or vectorization. The first step of our optimization process is to compute all valid partitionings of the program statements, such that a class of this partition corresponds to a set of *fusable statements*: those will share at least one common loop in the target code. Then, for each valid partitioning the second step is to apply model-driven optimizations individually on each class of the partition in a systematic fashion. These optimizations result in arbitrary compositions of affine loop transformations (skewing, interchange, multi-level distribution, fusion, peeling and shifting) to generate a loop nest for the class such that (1) first outer loop(s) are parallel and permutable; (2) data locality is maximized (this

is covered in Section V-A). Then, on the resulting loop nest a subsequent sequence of loop transformations is performed to expose parallel inner-loops with a minimal reuse distance to enable efficient vectorization (this is covered in Section V-B).

It is very hard to predict a profitable program partitioning, 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. This interplay is machine-specific, and to find a profitable partition for a program we will thus resort to iterative, feedback-directed search. On the other hand, profitability of optimizations such as tiling or vectorization are easier to assess, typically because they are generally beneficial if they do not destroy some other properties of the code, such as thread-level parallelism or data locality. We will thus rely on performance models for their selection.

### III. PROGRAM OPTIMIZATION

Most internal compiler 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 [37]. In addition, a rich algebraic structure is required when building complex compositions of such transformations [16], enabling efficient search space construction and traversal heuristics [26].

#### A. 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) [12], [16], 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 (constants that are unknown at compile time). Relaxation of these constraints based on affine over-approximations have been proposed [4]. Our optimization scheme is compatible with it, but we limit this work to describing the representation and optimization of regular SCoPs.

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 2mm kernel in Figure 1, the iteration domain of  $R$  is:

$$D_R = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i < N \wedge 0 \leq j < N\}$$

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

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  $e^{R \rightarrow S}$  is labeled with a dependence polyhedron  $\mathcal{D}_{R,S}$ , for all dependence polyhedra.

2) *Representing optimizations*: The second step of polyhedral program optimization is to compute 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 [16]. 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  $\vec{x}_S$  and program parameters  $\vec{n}$ . It is written

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

where  $T_S$  is a matrix of 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 [13], and that any loop transformation can be represented in the polyhedral representation [37].

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

$$\theta_i^S = [ t_1^S \ t_2^S \ \dots \ t_{m_S}^S \ t_0^S ]$$

That is,  $t_0^S$  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  $\Theta$  [16].

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 [3] to perform this task.

### B. Combined Iterative and Model-Driven 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 series of model-driven affine loop transformations: (1) a tiling-based optimization and parallelization algorithm; (2) a vectorization-based algorithm.

This scheme differs from previous work using iterative compilation to search for an affine multidimensional schedule [26], as we do not require anymore to empirically search for the full sequence of transformations. Instead, we limit the search to the most performance impacting part, aiming for a substantial reduction of the search space size while still preserving the most significant variability between candidate versions.

## IV. CONSTRUCTING VALID PARTITIONINGS

Enforcing that statements inside the same class can share at least one common loop is a high-level abstraction of loop fusion / distribution and code motion. If some statements in a given class were not fusable, then this partitioning would be equivalent to the one where the statements are distributed: this is a case of duplicate in the search space. Our approach to guarantee to find the most effective partitioning is to exhaustively evaluate all of them. In order to reduce to its minimum the time of the empirical search, it is thus needed to prevent the emergence of duplicates in the search space.

On the other hand, expressiveness is a major concern, as we aim at building a search space of *all valid* (that is, semantics-preserving) partitionings of the program. To achieve this goal we leverage the expressiveness of the polyhedral representation, and its capability to compute arbitrary enabling transformation for fusion. We first provide the broadest definition for fusion of statements in the polyhedral model in Section IV-A, before discussing the search space construction of all valid partitionings in Section IV-B.

### A. Fusion and fusability of statements

In the polyhedral model, loop fusion is characterized by the fine-grain interleaving of statement instances [6]. Two statements are fully distributed if the range of the timestamps associated to their instances never overlap. Syntactically, this results in distinct loops to traverse the iteration domains. One may define fusion as the negation of the distribution criterion. For such case we say that two statements  $R, S$  are fused under at least one common loop if there exists at least one pair of iterations for which  $R$  is scheduled before  $S$ , and another pair of iterations for which  $S$  is scheduled before  $R$ . This is stated in Definition 1.

*Definition 1 (Fusion of two statements)*: Given two statements  $R, S$ . They are fused at schedule level  $p$  if,  $\forall k \in \{1 \dots p\}$ , there exists at least two pairs of executed instances  $\vec{x}_R, \vec{x}_S$  and  $\vec{x}_R', \vec{x}_S'$  such that:

$$\Theta_k^R(\vec{x}_R) \leq \Theta_k^S(\vec{x}_S) \wedge \Theta_k^S(\vec{x}_S') \leq \Theta_k^R(\vec{x}_R')$$

But for fusion to have a performance impact, a stronger criterion is preferred to guarantee that at most  $c$  instances are

not finely interleaved. In general, computing the exact set of interleaved instances requires complex techniques. However this precision is not necessary to the success of our optimization algorithm, we thus allow for a lack of precision to present an easily computable test for fusability based on an *estimate* of the number of instances that are not finely interleaved [25].

*Definition 2 (Fusability):* Given two statements  $R, S$  such that  $R$  is surrounded by  $d^R$  loops, and  $S$  by  $d^S$  loops. They are fusable at schedule level  $p$  if,  $\forall k \in \{1 \dots p\}$ , there exist two semantics-preserving schedules  $\Theta_k^R$  and  $\Theta_k^S$  such that:

$$(i) \quad \forall k \in \{1, \dots, p\}, \quad |\Theta_k^R(\vec{0}) - \Theta_k^S(\vec{0})| < c$$

$$(ii) \quad \sum_{i=1}^{d^R} \theta_{k,i}^R > 0, \quad \sum_{i=1}^{d^S} \theta_{k,i}^S > 0$$

Condition (i) ensures that the number of iterations that are peeled from the loops is not greater than  $c$ : it implies the remaining iterations of  $R$  and  $S$  will be fused under a common loop.<sup>1</sup> Condition (ii) ensures that the schedule row  $k$  has non-null values for the coefficients attached to the loop iterators, that is, (ii) ensures that  $\Theta_k^R$  and  $\Theta_k^S$  are not constant schedules. This condition is required to guarantee that  $\Theta_k^R$  and  $\Theta_k^S$  represent a loop interleaving statement *instances* in the target code, and not simply a statement interleaving.

The only restriction on  $\Theta$  coefficients is  $\theta_{i,j} \in \mathbb{N}$ . Thus this definition takes into account any composition of loop interchange, skewing, multidimensional shifting, peeling and distribution that is required to fuse the statements under a common outer loop (possibly with prologue and epilogue).

To ensure that two statements are fusable, we resort to building a Parametric Integer Program [12] which contains sufficient constraints for the existence of a semantics-preserving multidimensional schedule [14], [25], in conjunction with the constraints given by Definition 2. If this program has a solution then the two statements are fusable.

## B. Modeling program partitionings

Our objective is to model a search space which contains all possible partitionings of a program, such that statements in the same class can be fused under a common outer loop. In addition, we also require the class identifier to reflect the order in which classes are executed, to model code motion. A general framework for this purpose has been developed by Pouchet [25] in the context of multi-level partitionings. However in the context of the present work we limit ourselves to the modeling of fusable statements at the outer loop level only, a restriction of the general case.

Returning to the 2mm example of Figure 3. The partitioning is  $\rho_{\text{papteron}} = \{\{R\}, \{T, S\}, \{U\}\}$ . We represent this partitioning using a vector representing the *statement interleaving* at the outer-most loop level. To reason about it, one may associate an identifier  $id^S$  to each statement  $S$  such that their ordering encodes exactly the ordering and fusion information for the outer loop level. Using this notation, one gets for  $\rho_{\text{papteron}}$  that

$\{id^R = 0, id^S = 1, id^T = 1, id^U = 2\}$ . This is noted  $\vec{f} = \begin{pmatrix} 0 & 1 & 1 & 2 \end{pmatrix}$ .

We explicitly **force**  $\vec{f}$  to exhibit important structural properties of the transformed loop nest:

- 1) if  $\vec{f}_i = \vec{f}_j$  then the statements  $i$  and  $j$  share (at least) 1 common loop;
- 2) if  $\vec{f}_i < \vec{f}_j$  then the statements  $i$  and  $j$  do not share any common loop, and  $i$  is executed before  $j$ .

However, intuitively, several choices for  $\vec{f}$  represent the same statement interleaving: for example, the transformed code is invariant to translation of all coefficients, or by multiplication of all coefficients by a non-negative constant. Consider the following example, for three statements  $R, S$  and  $T$ :

$$\vec{f} = \begin{pmatrix} 0 & 2 & 2 \end{pmatrix}$$

This ordering defines that  $S$  and  $T$  are fused together, and that  $R$  is not and is executed before  $S$  and  $T$ . An equivalent description is:

$$\vec{f}' = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

We observe that the number of duplicates using such a representation grows exponentially with the size of  $\vec{f}$ . As a major concern is to prevent duplicates in the space, we use an internal encoding for this problem such that the resulting space contains one and exactly one point per distinct partitioning [25].

The search space is modeled as a convex set of candidate partitionings, defined with affine inequalities. There are several motivating factors. The set of possible partitionings of a program is extremely large (in the order of  $10^{12}$  possibilities for 14 elements [31], with a super-exponential growth), while the space complexity of our convex set hardly depends on the cardinality of the set. Also, removing a subset of unwanted partitionings 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 [27], [26], and these techniques apply directly.

## C. Pruning for semantics preservation

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 [27], [26]. This is critical to allow any iterative search method to focus on relevant candidates only.

To remove all non-valid partitionings, we prune the space of all partitionings that does not allow to verify Definition 2 for all statements in the same class. Technically, we use an algorithm which iterates on possible partitionings and eliminates all invalid candidates, based on a graph representation of the problem [25]. To improve the speed of this algorithm, we also leverage some critical properties of fusability. The algorithm iterates on possible partitionings starting from the smallest size for the classes of the partition, leveraging that a superset of an unfusable set is not fusable. We also leverage a reduction of

<sup>1</sup>To ensure this statement holds true in all cases, further constraints to preprocess the iteration domains are needed [25]. They are omitted here for the sake of clarity and does not impact the applicability of this definition.

the problem of the transitivity of fusability to the computation of the existence of pairwise compatible loop permutations, as defined in [25]. In practice this algorithm proved to be very fast, and for instance computing all semantics-preserving interleavings at the first dimension takes less than 0.5 second for the benchmark `ludcmp`, pruning the set from about  $10^{12}$  possible partitionings to the remaining 20 valid ones.

## V. MODEL-DRIVEN OPTIMIZATIONS

Given a partitioning of the program statements, the second step is to perform aggressive, model-driven optimizations that respect this partitioning. We first discuss the computation of a sequence of loop transformations that implements the partitioning, and produce tiled parallel code when possible in Section V-A. We then present in Section V-B our approach for model-driven vectorization in the polyhedral model.

### A. Tiling hyperplanes

The first model-driven optimization we consider applies, individually on each class of the partition, a polyhedral transformation which implements the fusion of statements via a possibly complex composition of multi-dimensional tiling, fusion, skewing, interchange, shifting, and peeling. It is known as the Tiling Hyperplanes method [6], [7], that we *restrict to operate locally on classes of the partition*.

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 [6], [7]; 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 partitioning.

1) *Legality of tiling*: 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 condition, also known as forward communication only, makes it legal to execute the tile atomically. It is summarized in Definition 3

*Definition 3 (Legality of Tiling)*:

$$\Theta_d^S(\vec{x}_S) - \Theta_d^R(\vec{x}_R) \geq 0, \quad \langle x_R, x_S \rangle \in \mathcal{D}_{R,S} \quad (1)$$

Equation (1) must hold true for all dependences and all dimensions  $d$  to be tiled [18], [29], [6]. Selecting schedules such that each dimension is independent with respect to all others allows for a more efficient tiling. Rectangular or close to rectangular blocks are achieved when possible, avoiding complex loop bounds in the case of arbitrarily shaped tiles. We resort to augmenting the constraints, level-by-level, with orthogonality constraints [6].

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 V-A2. 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 loop level it is not possible to find legal hyperplanes for all statements, the statements are split [6], resulting in a loop distribution at this level. Note that by construction of valid partitionings this can not occur at the outer-most loop level.

2) *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 [6]. 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  $\vec{n}$ .

$$\mathbf{u} \cdot \vec{n} + w \geq \Theta_d^S(\vec{x}_S) - \Theta_d^R(\vec{x}_R) \quad \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S} \quad (2)$$

$$\mathbf{u} \in \mathbb{N}^p, w \in \mathbb{N}$$

The legality and bounding function constraints from (1) and (2) are recast through the affine form of the Farkas Lemma [13] such that the only unknowns left are the coefficients of  $\Theta_d$  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  $\Theta_d$ .

$$\text{minimize}_{\prec} (\mathbf{u}, w, \dots, t_{d,i}, \dots) \quad (3)$$

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 [6]. Because outer permutable bands have been exposed, multidimensional tiling can be applied on them.

3) *Profitability of the transformation*: The profitability of the Tiling Hyperplane method is complex to assess in its general formulation, as it is characterized by the profitability of loop fusion and the impact on subsequent vectorization. Our technique has removed the profitability estimate of outer-loop fusion/distribution, since we empirically evaluate all possible choices. In addition, we rely on a second stage dedicated to expose inner loops which are good vectorization candidates. So the problem of the profitability of the Tiling Hyperplane is reduced to the effectiveness of maximizing data locality in a given class, while outer-parallelism and vectorizable loops are made independent to the problem. Technically, one should consider the profitability of multi-level statement interleavings to guarantee that each possible loop structure is evaluated in order to find the best one, trading parallelism and locality at each loop level. However, focusing only on the outer-level carries the most important changes in parallelism and communication possibilities. In our optimization algorithm, we *chose to systematically apply the tiling hyperplane method on each class of the partition*.

4) *Applying tiling on a transformed loop nest*: Tiling a permutable loop nest is profitable in particular when there is reuse of data elements within the execution of a tile. Another criterion to take into account is to preserve enough iterations

at the inner-most loop level to allow for a profitable steady-state for vector operations within the execution of a tile. Our extremely simple algorithm to determine the tiling of a loop nest proceeds as follows:

- 1) compute the order of magnitude of data reuse in the loop nest;
- 2) compute the depth of the loop nest;
- 3) if there is  $O(N)$  reuse within a loop, and the loop nest depth is greater than 1 then tile the loop nest.

To achieve maximal performance it is expected that tuning the tile sizes can provide improvement, however the problem of computing the best tile sizes for a loop nest is beyond the scope of this paper. In our experiments tile sizes are computed such that data accessed by each tile roughly fits in the L1 cache.

### B. Vectorization

On modern, SIMD-capable architectures vectorization is a key for performance. The acceleration factor is a conjunction of several elements including, but not limited to, the number of elements packed in a vector and the throughput of the vector units. Making the most of SIMD units requires to succeed in the two following categories of optimizations:

- 1) high-level transformations, to expose parallel inner loops with maximally-aligned, minimally-strided accesses, which are good candidates for vectorization;
- 2) low-level transformations, to deal with hardware constraints such as realignment, vector packing or vector instruction selection.

In our framework we rely on the back-end compiler to produce *vectorized* code. A major challenge for production compilers is to detect and possibly transform the code to expose good vectorizable loops. They are geared towards the common case, and must provide extremely fast compilation time. Hence they lack the precision and expressiveness of the polyhedral framework, implementing instead approximate techniques. A consequence is that such compilers usually fail at computing a restructuring loop transformation to expose the best candidate for the inner-most loops, or may even fail to detect parallel inner-loops because of the complexity of the surrounding tile loop bounds. Moreover, they use general-purpose heuristics which may produce unvectorized code when dealing with arbitrary parametric loop bounds and array access alignments.

Our approach to vectorization leverages recent analytical modeling results by Trifunovic et al. [34]. We take advantage of the polyhedral representation to aggressively restructure the code, to expose vectorizable inner loops. High-level transformations and analytical modeling is performed in a very expressive framework, while deferring to the back-end compiler the task of performing low-level transformations. In addition we bypass the compiler’s high-level optimization stage by marking loops with `#pragma vector` and `#pragma ivdep`, when applicable.

1) *The Algorithm:* Our algorithm proceeds level-by-level, from the outer-most loop level to the inner-most. For each loop at that level, candidates for vectorization are computed

such that: (1) the loop can be moved to the inner-most position — via a sequence of loop interchanges — while preserving the semantics; and (2) moving this loop to the inner-most position does not remove thread-level parallelism. We then compute for each loop in the set of candidates, a cost metric based on the maximal distance (in memory) between data elements accessed by two consecutive iterations of this loop [34], considering all statements enclosed in this loop. The algorithm then moves to the next loop level, until all candidate loops for vectorization have been annotated with the cost metric. Loops with the best metric are then sunk inwards to the inner-most position, with a sequence of permutations captured within the polyhedral representation. The tiling hyperplane method guarantees that this sinking operation is always legal: this seamless coordination of the two methods is a key benefit of a polyhedral compilation framework. Note that because of parametric and possibly non-matching loop bounds, this transformation may result in additional prolog/epilog code surrounding the loops. This is handled seamlessly in the polyhedral representation but would have posed a major challenge to standard transformation frameworks.

2) *Profitability:* Combining the approach of Trifunovic et al. with the tiling hyperplane method leads to a very robust algorithm: it identifies the most profitable vectorization alternative in most cases.

Because we do not alter the general code structure (no subsequent distribution or parallelism removal), the profitability is only connected to sinking inwards a parallel loop that access data in a more contiguous fashion. When our algorithm fails at exposing the most profitable inner-most loop, the chosen loop is extremely likely to be a better candidate for vectorization than the one it replaces.

## VI. 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 available free software such as CLOOG, CLAN, CANDL, PIPLIB and POLYLIB.<sup>2</sup> 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 with ICC, intra-tile parallel loops are moved to the innermost position and marked with `ivdep` pragmas to facilitate compiler auto-vectorization, when possible.

### A. Experimental setup

We experimented on two high-end machines, representative of modern multi-core computing facilities: a 4-socket Intel hexa-core Xeon E7450 (Dunnington) at 2.4GHz with 64GB of memory (24 cores, 24 hardware threads) and a 4-socket AMD quad-core Opteron 8380 (Shanghai) at 2.50GHz (16

<sup>2</sup>A beta version of PoCC is available on request, a public release is planned in May.

	#loops	#stmts	#refs	#deps	#part.	#valid	Variability	Pb. Size
2mm	6	4	8	12	75	12	✓	1024x1024
3mm	9	6	12	19	4683	128	✓	1024x1024
adi	11	8	36	188	545835	1		1024x1024
atax	4	4	10	12	75	16	✓	8000x8000
bicg	3	4	10	10	75	26	✓	8000x8000
correl	5	6	12	14	4683	176	✓	500x500
covar	7	7	13	26	47293	96	✓	500x500
doitgen	5	3	7	8	13	4		128x128x128
gemm	3	2	6	6	3	2		1024x1024
gemver	7	4	19	13	75	8	✓	8000x8000
gesummv	2	5	15	17	541	44	✓	8000x8000
gramschmidt	6	7	17	34	47293	1		512x512
jacobi-2d	5	2	8	14	3	1		20x1024x1024
lu	4	2	7	10	3	1		1024x1024
ludcmp	9	15	40	188	10 <sup>12</sup>	20	✓	1024x1024
seidel	3	1	10	27	1	1		20x1024x1024

Fig. 6. Summary of the optimization process

cores, 16 hardware threads) with 64GB of memory. We also experimented on a potentially upcoming representative of low-cost computing platforms, a 1-socket Atom 230 processor at 1.6GHz with 1GB of memory (1 core, 2 hardware threads).

All systems were running Linux 2.6.x. We used ICC 11.1 with the options `-fast -parallel -openmp` referred to as `icc-par` and GCC 4.3.3 with options `-O3 -msse3 -fopenmp` as `gcc`. For the Atom we used `-march=prescott -mtune=pentium -funroll-loops` as additional GCC flags, those are reported as giving the best performance for this processor.

We consider 16 benchmarks, typical from compute-intensive sequences of algebra operations. 2mm, 3mm, atax, bicg, gemm, gemver, gesummv, are (sequences) of linear algebra operations, on vectors and matrices. covar and correl are covariance and correlation computations for use in Principal Component Analysis (sources: StatLib). adi, 2d-jacobi and seidel are stencil computations. Finally doitgen, gramschmidt, lu and ludcmp are solver and manipulation algorithms operating on matrices. All these benchmarks are part of the PolyBenchs test suite [24] and are freely available for download.

## B. Summary of experiments

Figure 6 presents the main characteristics of our benchmark suite. We report, for each benchmark, some information on the considered SCoP (#loops the number of loops, #stmts the number of statements, #refs the number of array references, #deps the number of dependences). We report also the number of possible (including invalid) partitionings as #part., and the number of semantics-preserving partitionings #valid to highlight the pruning factor coming from our algorithm. We also check the column Variability each time we observed a 5% or more difference between the best versions found for a platform and its execution on the other platforms, this to emphasize the requirement for a tuning of the partitioning selection. Finally, we also report the dataset size we used for the benchmarks (Pb. Size).

## C. Detailed performance evaluation

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 optimization process for all presented benchmarks took less than one hour on the slowest machine. This time is totally dominated by the execution time of each candidate, had we used a smaller/larger dataset sizes the optimization time would have decreased/increased.

1) *Performance improvement*: We report in Figure 7 the performance improvement of our technique when compared to the native production compiler with aggressive optimization flags enabled used as the baseline (performance improvement = 1). To study in a fair fashion the benefit of our methods, we particularly study two specific partitionings:

- *maxfuse*, which corresponds to applying the tiling hyper-plane method on the full program instead of locally to each class of the partition;
- *smartfuse*, which corresponds to a partitioning where statements that do not share any data reuse are put in different classes. This is considered as the state-of-the-art [6].

To further emphasize the benefits of our approach, we report the performance improvement of smart fusion when used without our complementary step for vectorization (`pluto-smartfuse`), and with it (`pocc-smartfuse`). The best performance improvement found by our combined approach is reported in iterative.

We obtain solid performance improvements over the native compiler, above 2× in average better for Xeon and 2.5× for Opteron. For the case of Atom, we observed that GCC fails in many situations to exhibit both coarse-grain and fine-grain parallelism in the input code, this leads to very large improvements by our framework: up to 30× for matrix multiplications.

For most benchmarks, smart fusion is performing better than maximal fusion, showing the importance of controlling the cache pressure and exposing enough inner-parallel loops. A model-driven approach such as maxfuse which looks for the minimization of synchronizations and maximization of locality is still likely to provide a performance improvement over general-purpose heuristics implemented in production compiler, as shown in Figure 7. Such examples are shown for the benchmarks where there is only one possible partitioning, thus equivalent to applying maxfuse to the full program. However, empirical search is needed for 9 out of 16 benchmarks to exhibit the best performance, for a benefit of up to 2× over

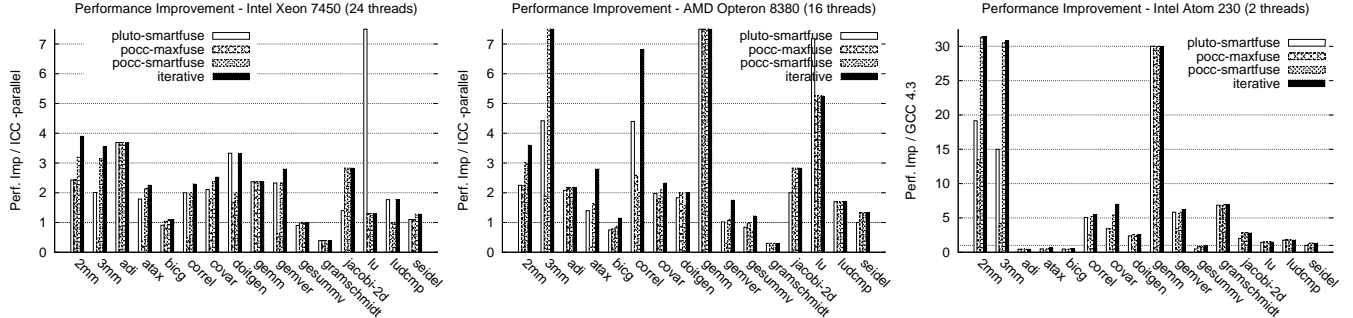


Fig. 7. Performance improvement of (a) state-of-the-art smart fusion without dedicated vectorization stage (pluto-smartfuse); (b) two specific partitionings of our search space: maximal fusion and smart fusion, both with dedicated vectorization stage (pocc-maxfuse and pocc-smartfuse); and (c) the best found partitioning after empirical search (iterative). Baseline is ICC -fast -parallel -openmp for Xeon and Opteron, GCC -O3 -mssse3 -march=prescott -mtune=pentium -funroll-loops for Atom.

smart fusion.

2) *Performance portability*: Beyond absolute performance improvement, another motivating factor for iterative selection of partitionings 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 8 the relative performance normalized with respect to icc-par on Opteron of gemver, for the Xeon and Opteron architectures. The version index is plotted on the x axis, 1 is max-fuse and 8 is maximal distribution.

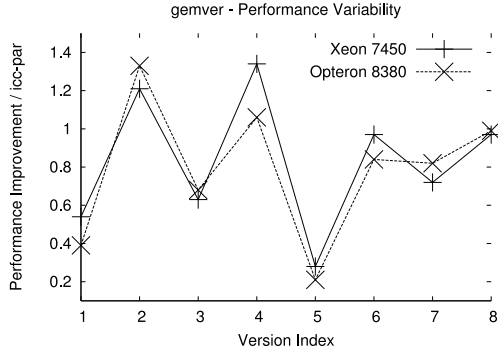


Fig. 8. Performance variability for gemver

For Xeon, the best version is 4, which corresponds to the partitioning (0,0,1,2). It performs 10% better than version 2, which corresponds to the partitioning (0,0,0,1). Interestingly, for Opteron, version 4 performs 20% slower than version 2.

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 partition-

ing, whatever the specifics of the program, compiler and architecture.

## VII. RELATED WORK

Traditional work on loop fusion [20], [22], [23], [30] are restricted in their ability to find complex partitionings. 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 [23] 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. [11], [10] 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 [36], [28]. 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<sup>3</sup> and IBM XL.

Bondhugula et al. designed Pluto, the first integrated fusion and tiling heuristic based on the polyhedral model [6], [7], which subsumes a large set of additional loop transformations (interchange, skewing, shifting). It inherits the flexibility of the tiling hyperplane method [18], [17] 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 [19], [9], [16], [8],

<sup>3</sup>Graphite development branch: <http://gcc.gnu.org/wiki/Graphite>, and now in GCC 4.5.

[33]. They capture partitionings, 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 [5], [32], [1], [21], [28], [15], [26], [35]. 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.

### VIII. 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 combined iterative and model-driven approach, leveraging a state-of-the-art parallelization method based on loop tiling, and combining it with a novel 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 24-core Xeon, a 16-core Opteron, and a single-core low-power Atom processor. 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 performance improvements over the state-of-the-art model-based compilers, with performance improvement factors up to  $8\times$  over Intel'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.

*Acknowledgments:* This work was supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, the U.S. National Science Foundation through awards 0926687/0926688, and by the Army through contract W911NF-10-1-0004.

### 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] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.
- [3] 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.
- [4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, pages 283–303, Paphos, Cyprus, Mar. 2010.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] A. Darte and G. Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.
- [11] 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.
- [12] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.
- [13] 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.
- [14] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] F. Irigoien and R. Triolet. Supernode partitioning. In *ACM SIGPLAN Principles of Programming Languages*, pages 319–329, 1988.
- [19] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.
- [24] PolyBenchs 1.0. Available at <http://www-rocq.inria.fr/pouchet/software/polybenchs>.
- [25] L.-N. Pouchet. *Iterative Optimization in the Polyhedral Model*. PhD thesis, INRIA Saclay and University of Paris-Sud 11, Jan. 2010.
- [26] 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.

- [27] 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.
- [28] 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.
- [29] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.
- [30] S. Singhai and K. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.
- [31] N. J. A. Sloane. Sequence a000670. The On-Line Encyclopedia of Integer Sequences, <http://www.research.att.com/~njas/sequences/A000670>.
- [32] 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.
- [33] 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.
- [34] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.
- [35] 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.
- [36] 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.
- [37] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.