Predictive Modeling in a Polyhedral Optimization Space

Eunjung Park*, Louis-Noël Pouchet†, John Cavazos*, Albert Cohen‡ and P. Sadayappan†
* University of Delaware
{epark,cavazos}@cis.udel.edu
† The Ohio State University
{pouchet,saday}@cse.ohio-state.edu
‡ INRIA Saclay – Île-de-France
albert.cohen@inria.fr

***Abstract*—Significant advances in compiler optimization have been made in recent years, enabling many transformations such as tiling, fusion, parallelization and vectorization on imperfectly nested loops. Nevertheless, the problem of finding the best combination of loop transformations remains a major challenge. Polyhedral models for compiler optimization have demonstrated strong potential for enhancing program performance, in particular for compute-intensive applications. But existing static cost models to optimize polyhedral transformations have significant limitations, and iterative compilation has become a very promising alternative to these models to find the most effective transformations. But since the number of polyhedral optimization alternatives can be enormous, it is often impractical to iterate over a significant fraction of the entire space of polyhedrally transformed variants. Recent research has focused on iterating over this search space either with manually-constructed heuristics or with automatic but very expensive search algorithms (e.g., genetic algorithms) that can eventually find good points in the polyhedral space.**

In this paper, we propose the use of machine learning to address the problem of selecting the best polyhedral optimizations. We show that these models can quickly find high-performance program variants in the polyhedral space, without resorting to extensive empirical search. We introduce models that take as input a characterization of a program based on its dynamic behavior, and predict the performance of aggressive high-level polyhedral transformations that includes tiling, parallelization and vectorization. We allow for a minimal empirical search on the target machine, discovering on average 83% of the search-space-optimal combinations in at most 5 runs. Our end-to-end framework is validated using numerous benchmarks on two multi-core platforms.

I. Introduction

A significant amount of the computation time in scientific and engineering applications is usually spent in loops, making high-level loop transformations critical to achieving high performance for a variety of programs. The best loop optimization sequence is often not only program-specific, but also depends on the target hardware. Pouchet et al. illustrated this by showing the critical impact of tuning polyhedral optimizations for obtaining the best performance for a variety of numerical programs on different target processors [31], [30], [32].

Although significant advances have been made in developing advanced compiler optimization and code transformation frameworks, it remains an extremely challenging problem to find the best sequence(s) of high-level loop transformations to optimize performance on a given architecture. Existing static models are limited to highly simplified execution models of the machine. The very complex interplay between all the hardware resources (e.g., different cores with multiple levels of private/shared cache and TLBs, instruction pipelines, hardware pre-fetchers, SIMD units etc.) makes it extremely difficult to construct a static model that can accurately predict the effectiveness of a given set of loop transformations. Worse still, some optimization strategies may have conflicting objectives: for example, maximizing thread-level parallelism may inhibit SIMD-level parallelism, and may also result in degradation of data locality and increased memory footprint.

In the quest for higher and more portable performance, the compiler community has explored research based on iterative compilation and machine learning to *tune the compiler optimization flags or pass sequence*, to find the best (ordered) set for a given combination of benchmarks and target architectures. Although significant performance improvements have been demonstrated [26], [1], [17], [28], the performance obtained has generally been limited by the optimizations selected for automatic tuning, and by the degrees of freedom available for exploration. We identify two main limitations of iterative compilation efforts so far. First, most compilers lack a powerful high-level optimization framework: to date aggressive optimizations such as parallelization or vectorization as implemented in production compilers may simply fail to restructure the code enough to expose good parallel or vector loops. Second, the support for loop tiling (also called loop blocking) in production compilers is quite limited: usually able to tile only some perfectly nested loops, without any pre-transformation capabilities to help expose tiling opportunities, and almost no support for tuning the tile sizes. This is a critical performance issue as tiling is often the key loop transformation to achieve good data locality and parallelization [41].

The *polyhedral optimization framework* has been demonstrated as a powerful alternative to abstract-syntax-tree based loop transformations. The polyhedral framework enables the modeling of an arbitrarily complex sequence of loop transformations in a single optimization step. The downside of this expressiveness is the extreme difficulty of selecting an *effective* set of affine transformation coefficients that result in good performance, combining tiling, coarse and fine grain parallelization, together with fusion, distribution, interchange, skewing, permutation and shifting [18], [30], [32].

Previous work on iterative compilation based on this model showed that there is opportunity for very large performance improvement over native compilers [31], [30], significantly better than using standard compilation flag tuning or pass selection and ordering. However, directly tuning the polyhedral transformation in its original abstract representation remains a highly complex problem, where the search space is usually infinite. Despite progress in understanding the structure of this space and how to bound its size, this problem remains hardly tractable in its original form.

Past and current work in polyhedral compilation has contributed algorithms and tools to expose model-driven approaches for various high-level transformations, including:

- loop fusion and distribution, to partition the program into independent loop nests;
- loop tiling, to partition (a sequence of) loop nests into blocks of computations;
- thread-level parallelism extraction;
- SIMD-level parallelism extraction.

Bondhugula et al. proposed the first integrated heuristic for parallelization, fusion and tiling in the polyhedral model [4], [5], subsuming all the above optimizations into a single, tunable cost-model. Individual objectives such as the degree of fusion or the application of tiling can implicitly be tuned by minor ad-hoc modifications of Bondhugula's cost model.

We now summarize the contributions of the present paper. We address the problem of effectively balancing the trade-off between all the aforementioned high-level optimizations to achieve the best performance. As a direct benefit of our problem formalization, *we integrate the power of iterative compilation schemes with the expressiveness and efficiency of high-level polyhedral transformations*. Our technique relies on a training phase where numerous possibilities to drive the high-level optimizer are tested, using a source-to-source polyhedral compiler on top of a standard production compiler. We show how the problem of selecting the best optimization criteria can be effectively learned using feedback from the dynamic behavior of various possible high-level transformations. By correlating hardware performance counters to the success of a polyhedral optimization sequence, *we are able to build a model that predicts very effective polyhedral optimization sequences for an unseen program*. Our results show it is possible to achieve close to the search-space-optimal performance by testing no more than 5 program versions. To the best of our knowledge, this is the first effort that demonstrates very effective discovery of complex high-level loop transformations using machine learning models.

In Section II, we first present details on the optimization space we consider, before presenting the machine learning approach in Section III. Experimental results are presented in Section IV. We discuss related work in Section V.

II. Optimization Space

High-level loop transformations are crucial to effectively map a computation onto the target hardware. Effective mapping typically requires the partitioning of the computation into disjoint parts to be executed on different cores, and the transformation of those partitions into streams to be executed on each SIMD unit. In addition, the data flow used by the computation may need to be reorganized to better exploit the cache memory hierarchy and improve communication cost.

Addressing these challenges for compute-intensive programs has been demonstrated to be a strength of the polyhedral optimization framework. Several previous studies have shown how tiling, parallelization, vectorization or data locality enhancement can be efficiently addressed in an affine transformation framework [21], [34], [14], [24], [36]. Any loop transformation can be represented in the polyhedral representation, and composing arbitrarily complex sequences of loop transformations is seamlessly handled by the framework. This expressiveness and ease in composing and applying transformations are the strengths of the polyhedral model. However, the space of possible optimizations is dramatically enlarged, imposing challenges on the selection algorithms. In contrast to previous iterative approaches requiring the evaluation of up to hundreds of possible transformations [30], [32], we develop here a scheme that requires at most 5 candidate choices to be evaluated.

We also observe that high-level transformations are by far not sufficient to achieve the best performance for a program. Numerous low-level optimizations are required, and chip makers such as Intel have developed extremely efficient closed-source compilers for their processors. Unfortunately we have to consider such compilers as black-boxes, because of the difficulty in precisely determining which optimizations is implemented and when. Our approach considers the back-end compiler as part of the target machine, and we focus exclusively on driving the optimization process via high-level source-to-source polyhedral transformations.

We next present the set of optimizations that we consider.

A. Polyhedral Model

Sequences of (possibly imperfectly nested) loops amenable to polyhedral optimization are called *static control parts* (SCoP) [14], [18], roughly defined as a set of (possibly imperfectly nested) consecutive statements such that all loop bounds and conditionals are affine functions of the surrounding loop iterators and global variables (constants that are unknown at compile time but invariant in the loop nest). Relaxation of these constraints to arbitrary side-effect free programs has recently been proposed [3], and our optimization scheme is fully compatible with this extended polyhedral model.

Polyhedral program optimization involves the analysis of the input program to extract its *polyhedral representation*, including dependence information and array access patterns. These are defined at the granularity of the statement instance, that is, an executed occurrence of a syntactic statement.

A program transformation is represented by an affine multidimensional schedule. This schedule specifies the order in which each statement instance is executed. A schedule captures in a single step what may typically correspond to a sequence of several textbook loop transformations [18]. Arbitrary compositions of affine loop transformations (e.g., skewing, interchange, multi-level distribution, fusion, peeling and shifting) are embedded in a single affine schedule for the program. Every static control program has a multidimensional affine schedule [14], and tiling can be applied by extending the iteration domain of the statements with additional tile loop dimensions, in conjunction with suitable modifications of the schedule [18].

Finally, syntactic code is regenerated 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.

B. Polyhedral Optimizations Considered

High-level optimization primitives, such as tiling or parallelization, often require a complex sequence of enabling loop transformations to be applied while preserving the semantics. As an example, tiling a loop nest may require skewing, fusion, peeling and shifting of loop iterations before it can be applied. A limitation of previous approaches, whether polyhedral-based [25], [31] or syntactic-based [8], was the challenge of assessing the impact of the main optimization primitives, since the *enabling* sequence also had to be considered. This led most previous work to be limited in applicability: the enabling transformations were not considered in an integrated fashion, so that transformations such as tiling and coarse-grained parallelization could not be applied in the most effective fashion on numerous programs.

We address this issue by decoupling the problem of selecting a polyhedral optimization into two steps: (1) select a sequence of high-level primitives in the set $\{$ fusion/distribution, tiling, parallelization, vectorization, unroll-and-jam $\}$, this selection being based on machine learning and feedback from hardware performance counters, and (2) for the selected high-level primitives, use *static cost models* to compute the appropriate enabling transformations that implement the given sequence of high-level primitives. We thus keep the expressiveness and applicability of the polyhedral model, while focusing the selection decision only on the main transformations.

1) Loop Tiling: Tiling is a crucial loop transformation for parallelism and locality. It partitions the computation into blocks that can be executed atomically. When tiling is chosen to be applied on a program, we rely on the Tiling Hyperplane method [5] to compute a sequence of enabling loop transformations to make tiling legal on the generated loop nests.

Two important performance factors must be considered for the profitability of tiling. Tiling may be detrimental as it may introduce complex loop structure and the computation overhead may not be compensated by the locality improvement. This is particularly the case for computations where data locality is not the performance bottleneck. Second, the *size* of the tiles could have a dramatic impact on the performance of the generated code. To obtain good performance with tiling, the data footprint of an atomic tile should typically reside in the L1 cache. The problem of selecting the optimal tile size is known to be very hard and empirical search is often used for high-performance codes [40], [42], [37]. To limit the search space while preserving significant expressiveness, we allow the specification of a limited number of tile sizes to be considered for *each tiled loop*. In our experiments, we use only two possible sizes for a tile dimension: either 1 (i.e., no tiling along this loop level) or 32. The total number of possibilities is a function of the depth of the loop nest to be tiled: for instance, for a doubly-nested loop we test rectangular tiles of size 1×1 (no tiling), 1×32, 32×1 and 32×32.

2) Loop Fusion/Distribution: In the framework used in the present paper, there is an equivalence between (i) maximally fusing statements, (ii) maximizing the number of tilable loop levels, (iii) maximizing locality and (iv) minimizing communications. In its seminal formulation, Bondhugula proposed to find a transformation that maximizes the number of fused statements on the whole program using an Integer Linear Program encoding of the problem [4]. However, maximally fusing statements may prevent parallelization and vectorization, and the trade-off between improving locality despite reducing parallelization possibilities is not captured. Secondly, fusion may interfere with hardware prefetching. Also, after fusion, too many data spaces may contend for use of the same cache, reducing the effective cache capacity for each statement. Conflict misses are also likely to increase. Obviously, systematically distributing all loops is generally not a better solution as it may be detrimental to locality.

The best approach clearly depends on the target architecture, and the performance variability of an optimizing transformation across different architectures creates a burden in devising portable optimization schemes. We consider in this paper three high-level fusion schemes for the program: (1) NoFuse, where we do not fuse at all; (2) SmartFuse, where we only fuse together statements that carry data reuse; and (3) MaxFuse, where we try to maximally fuse statements. These three cases are easily implemented in the polyhedral framework, simply by restricting the cost function of the Tiling Hyperplane method to operate only on a given (possibly empty) set of statements.

Interaction with tiling: The scope of application of tiling directly depends on the fusion scheme applied on the program. Only statements under a common outer loop may be grouped in a single tile. Maximal fusion results in tiles performing more computations, while smart fusion may result in more tiles to be executed but with fewer operations in them. The cache pressure is thus directly driven by the fusion and tiling scheme.

3) Thread-level parallelization: Thread-level parallelism is not always beneficial, e.g., with small kernels that execute few iterations or when it prevents vectorization.

When a loop nest is tiled, it is always possible to execute the tiles either in parallel or in a pipeline-parallel fashion. For untiled loops, we rely on a cost model that pushes dependences to the inner loop levels, naturally exposing outer parallel loops. To drive thread-level parallelization we expose two options: (1) Parallel where we use OpenMP and insert a `#pragma omp parallel for` above the outer-most parallel loop of each loop nest; and (2) NoParallel where no pragma is inserted and no transformation is performed.

4) SIMD-level parallelization: Our approach to vectorization leverages recent analytical modeling results by Trifunovic et al. [36]. We take advantage of the polyhedral representation to restructure imperfectly nested programs, to expose vectorizable inner loops. The most important part of the transformation to enable vectorization comes from the selection of which parallel loop is moved to the innermost position. The cost model selects a synchronization-free loop that minimizes the memory stride of the data accessed by two contiguous iterations of the loop [36]. Note, this interchange may not always lead to the optimal vectorization, or may simply be useless for a machine which does not support SIMD instruction. We expose two options: (1) Vector, where the schedule is modified to expose good (ie, stride one memory accesses) vectorizable innermost loops, which are marked with `ivdep` and `vector always` pragmas to facilitate compiler auto-vectorization; and (2) NoVector, where no additional transformations are performed to enable vectorization.

5) Loop unroll-and-jam: Loop unrolling is known to help expose instruction-level parallelism. Tuning the unrolling factor can influence register pressure, in a manner that is compiler and machine-dependent. We expose three options that are applied on all innermost loops of the program: (1) NoUnroll; (2) UnrollBy4; and (3) UnrollBy8.

C. Putting it all Together

1) Generating the Final Transformation: A sequence of high-level primitives is encoded as a fixed-length vector of bits, referred to as T. To each distinct value of T corresponds a distinct combination of the above primitives. Technically, on/off primitives (i.e., Tile/NoTile, Parallel/NoParallel, etc.) are encoded using a single bit. Non-binary primitives such as the unroll factor or the tile sizes are encoded using a "thermometer" scale. As an illustration, to model unroll-and-jam factors we use two binary variables (x,y). The pair $(0,0)$ denotes no unroll-and-jam, an unroll factor of 4 is denoted by $(0,1)$ and unroll factor of 8 by $(1,1)$. Different tile sizes are encoded in a similar fashion. In our experiments, we only model the tile size on the first three dimensions (leading to 9 possibilities), and use a constant size for T. Thus for programs where the tiles have a lower dimensionality, some bits in T have no impact on the transformation.

To generate the polyhedral transformation corresponding to a specific value of T, we proceed as follows.

1) Partition the set of statements according to the fusion choice (one in NoFuse, SmartFuse or MaxFuse);
2) Apply the Tiling Hyperplane method [5] locally on each partition to obtain a schedule for the program that (a) implements the fusion choice, (b) maximizes the number of parallel loops, (c) maximizes the number of tilable dimensions [4] on each individual partition;
3) Modify the schedule according to the vectorization cost model, if Vector is set, to expose inner parallel loops;
4) Tile all tilable loop nests, if any, if Tile is set. The tile sizes to be used are encoded in T.

Other transformations do not require further modification of the program schedule. Depending on their activation in T, they are applied as post-pass on the generated program, as they only require syntactic modifications to the code (e.g., inserting pragmas or unrolling the code).

2) Candidate Search Space: The final search space we consider depends on the program. For instance, not all programs exhibit coarse-grain parallelism or are tilable. For cases where a primitive has no effect on the final program because of semantic considerations, multiple values of T lead to the same candidate code version. The search space, considering only values of T leading to distinct transformed programs, ranges from 91 to 432 in our experiments, out of 864 possible combinations that can be encoded in T.

III. Selecting Effective Transformations

Since we have removed the problem of computing enabling transformations, we can focus the search on the primitives with the highest impact as described in Section II. When considering a space of semantics-preserving polyhedral optimizations, even the most aggressive bounding can lead to billions of possible polyhedral optimizations [30]. We achieved a tremendous reduction in the search space size when compared to these methods, but we have hundreds of sequences to consider. In this paper, we propose to formulate the selection of the best sequence as a learning problem, and use off-line training to build predictors that compute the best sequence(s) of polyhedral primitives to apply to a new program.

A. Characterization of Input Programs

We focus in this work on the *dynamic* behavior of programs, by means of hardware performance counters. Using those abstracts away the specifics of the machine, and overcomes the lack of precision of static performance models. Also, models using performance counter characteristics of programs have been shown to out-perform models that use only static features of program [8].

A given input program is represented by a feature vector of performance counters collected by running the full program on the machine. We use the PAPI library [27] to gather information about memory management, vectorization and processor activity. In particular, for all cache levels and TLB levels we collect the total number of accesses and misses, the total number of stall cycles, the total number of vector instructions, and the total number of issued instructions. All counter values are normalized using the total number of instructions of the program.

B. Speedup Prediction Model

A general formulation of the optimization problem is to construct a search function that takes as input features of a program being optimized and generates as output one or more optimization sequences predicted to perform well on that program. Previous work [13], [8] has proposed to model the optimization problem by characterizing a program using performance counters. We use a prediction model originally proposed by Cavazos *et al.* [12], [7], but slightly adapted to support polyhedral primitives instead. We refer to it as a *speedup predictor model*.

This model takes as an input a tuple (F,T) where F is the feature vector of all hardware counters collected when running the original program; and T is one of the possible sequence of polyhedral primitives. Its output is a prediction of the speedup T should achieve, relative to the performance of the original code. Figure 1 illustrates the speedup prediction model. For a given input program, first the feature vector of performance counters are collected. Then, the model is ask to predict the expected speedup of a primitive sequence T. By predicting the performance of each possible sequence, it is possible to rank them according to their expected speedup and select the sequence(s) with the highest speedup.

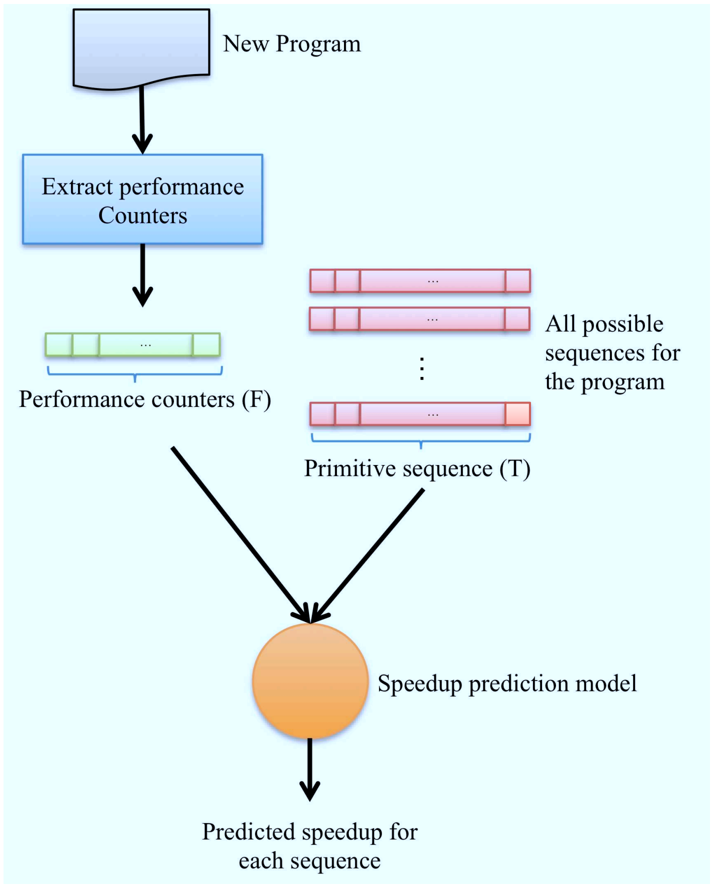
New Program
Extract performance Counters
...
Performance counters (F)
All possible sequences for the program
Primitive sequence (T)
Speedup prediction model
Predicted speedup for each sequence

Fig. 1. Speedup prediction model

We implemented the speedup prediction model by using two different machine learning algorithms – regression and SVM (Support Vector Machine), using Weka [6].

The Regression based model demonstrates the relationship between dependent and independent variables, and we can use this model to expect dependent variables according to the change of given independent variables. We used linear regression to fit the predictive model to dependent variable which is speedup of programs, and independent variables which are performance counters and the polyhedral optimization sequence. Regression model often makes assumptions about the data-generating process, and this is often useful for prediction even though the assumption is not correct. However, regression model may not be optimal because of this and possibly mislead results when we use incorrect assumptions.

SVM is a supervised machine learning technique, used for both classification and regression, and it can apply linear techniques to non-linear problems. First, SVM transforms data into a linear space by using kernel functions, and uses a linear classifier to separate data with a hyperplane. SVM not only finds a hyperplane to separate data, but also finds the best hyperplane, so called maximum margin hyperplane, showing the largest separation from the set of hyperplanes.

C. Model Generation and Evaluation

We train a specific model for each target architecture, as the specifics of a machine (e.g., cache miss cost, number of cores, etc.) significantly influence what transformations are effective for it. In addition, to evaluate the quality of linear regression versus SVM, we train one specific model for each.

A model is trained as follows. For a given program P in the training set, (1) compute its execution time E and collect its performance counters F; (2) for all possible sequences of polyhedral primitives T_i, apply the transformation to P and execute the transformed program on the target machine, this gives an execution time E_{T_i}, and the associated speedup $S_{T_i} = E/E_{T_i}$; (3) train the model with the entry $(F,T_i) = S_{T_i}$. This is repeated for all programs in the training set. This is illustrated in Figure 2.

Each of our models must predict optimizations to apply to unseen programs that were not used in training the model. To do this, we need to feed as input to our models a characterization of the unseen program. We then ask the model to predict the speedup of each possible transformations sequences T_i in our optimization space, given the unseen program characteristics. We order the predicted speedups to determine which sequence is predicted best, and apply it to the unseen program.

Note in the experiments presented below, we use the standard *Leave One Out Cross-Validation* procedure for evaluating our models. That is, the two models (SVM or LR) are trained on $N-1$ benchmarks, and evaluated on the benchmark that has been left out. This procedure is repeated individually for each benchmark to be evaluated: each evaluation is done on a program that was never seen by the model during the training.

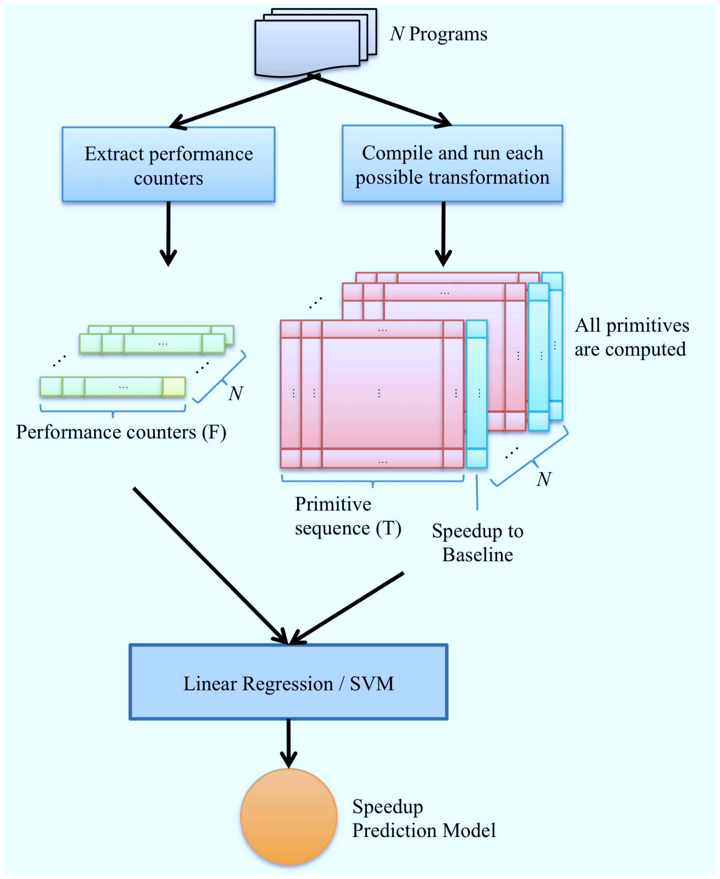

Fig. 2. Overview of the training phase

D. One-shot and Multi-shot Evaluation

The models presented above output a single optimization sequence for an unseen program. For the rest of the paper, we refer to this approach as the 1-shot model.

It is worth considering an empirical evaluation of several candidate transformations, as the predictor may not predict correctly the actual best sequence for the program. A typical source for misprediction comes from the back-end compiler: depending on the input *source* code, it may performs specific optimizations based on pattern-matching, for instance. As an illustration, we observed in our experiments that for the benchmark 2mm (computing two matrix multiplications $tmp = A.B; output = tmp.C$), the best performance when using Intel ICC 11.0 is achieved when *no tiling* is applied by our framework, despite high cache miss ratios. We suspect this is because ICC performs specific optimizations on this particular computation (matrix-multiply), since in this setup tiling 2mm to make it L1-resident decreases the performance. However, another program with similar hardware counter features may be processed entirely differently by ICC, and as shown by our experiments even the same program is handled differently by ICC 11.0 and ICC 11.1 on two different machines.

We propose to evaluate also 2-shot and 5-shot models. For the 2-shot model, we keep the two predicted best sequences, apply each of them and execute both transformed programs on the machine; we then keep the one that in practice performs best. This implies iteratively testing two candidate transformations. Similarly, we end up testing on the machine five candidate transformations with the 5-shot models.

IV. Experimental Results

A. Experimental Setup

We provide experimental results on two multi-core systems: Nehalem, a 2-socket 4-cores Intel Xeon 5620 (16 H/W threads), and R900, a 4-socket 6-cores Intel Xeon E7450 (24 H/W threads). Both systems have 16 GB of memory and run Linux. The back-end compiler used for the baseline and all candidate polyhedral optimizations is Intel ICC with option `-fast`, version 11.1 for Nehalem and version 11.0 for R900.

Our benchmark suite is PolyBench v2 [20], composed of 28 different kernels and applications containing static control parts. The datasets are the reference one [20], and most benchmarks are L3-resident in our testing framework.

B. Comparison of LR, SVM and Random

We show in Table I-II the performance of the three different models we have evaluated. For each benchmark, we report the performance improvement over the original code, when compiled with `icc -fast`, for the 1-shot, 2-shot and 5-shot approaches. In particular we report LR for Linear Regression, SVM for Support Vector Machine, R for Random (averaging 100 experiments) and %Opt the fraction of the optimal performance improvement achieved by the best of LR and SVM. Regarding the search space, we report Opt the best improvement achieved by a candidate optimization in our search space. We also report in the column Poly the performance improvement achieved when using a polyhedral static cost model to select the transformation [4]. In our nomenclature, it corresponds to MaxFuse and Parallel and Tile, using a default tile size of 32 in each tiled dimension. We also compare against a tuning of 12 flag optimization sequences for ICC (one of `-O2`,`-O3`,`-fast`, with and without `-parallel` to turn on and off automatic parallelization, and with and without `-no-vec` to turn on and off vectorization). We report the improvement achieved by the best flag sequence applied on the original code in the ICC column.

Analysis: First, we observe that polyhedral optimization tuning significantly outperforms ICC flag tuning, from 2$\times$ to 3.5$\times$ better performance is achieved on average. And for all benchmarks and all architectures, there exists at least one polyhedral sequence which outperforms ICC. We also observe that the polyhedral static cost model we use for comparison is significantly outperformed by our approach. This static model has proved its effectiveness for programs with significant data reuse, as in correlation and covariance for instance. Nevertheless, for numerous programs tiling and/or parallelization is detrimental to performance, as in gesummv or dynprog. The performance drop mainly comes from the very complex loop structure that is generated with polyhedral tiling, which in turn inhibits numerous scalar optimizations on

TABLE I
PERFORMANCE IMPROVEMENTS FOR INTEL XEON E5620 (BASELINE: ICC 11.1 -FAST)

				1-shot				2-shot				5-shot			
Benchmark	Opt	Poly	ICC	LR	SVM	R	%Opt	LR	SVM	R	%Opt	LR	SVM	R	%Opt
2mm	13.8×	4.07×	1.00×	13.8×	13.8×	2.67×	100%	13.8×	13.8×	3.87×	100%	13.8×	13.8×	5.28×	100%
3mm	11.9×	2.17×	1.00×	2.46×	0.81×	1.54×	20.67%	2.46×	2.33×	2.53×	20.67%	2.46×	2.33×	3.71×	20.67%
adi	3.73×	3.66×	1.86×	3.22×	3.22×	1.30×	86.33%	3.22×	3.22×	2.32×	86.33%	3.22×	3.22×	2.82×	86.33%
atax	2.40×	2.00×	1.31×	0.85×	2.39×	1.14×	99.58%	0.85×	2.39×	1.30×	99.58%	1.41×	2.40×	1.70×	100%
bicg	1.61×	0.75×	1.27×	0.59×	0.58×	0.49×	36.65%	0.59×	0.58×	0.81×	36.65%	0.59×	1.61×	1.00×	100%
cholesky	1.00×	0.88×	1.00×	0.41×	0.97×	0.55×	97.98%	0.41×	0.97×	0.80×	97.98%	0.41×	0.97×	0.94×	97.98%
correlation	21.1×	2.88×	3.24×	8.98×	10.7×	4.30×	50.81%	8.98×	10.7×	6.43×	50.81%	11.8×	17.8×	10.2×	84.71%
covariance	21.5×	13.0×	3.25×	21.5×	21.5×	5.29×	100%	21.5×	21.5×	5.90×	100%	21.5×	21.5×	9.72×	100%
doitgen	12.5×	4.15×	1.00×	1.06×	3.67×	2.15×	29.34%	1.06×	3.85×	3.23×	30.78%	3.39×	3.95×	5.09×	31.60%
durbin	1.00×	1.00×	1.00×	0.99×	1.00×	0.99×	100%	0.99×	1.00×	1.00×	100%	0.99×	1.00×	1.00×	100%
dynprog	1.01×	0.32×	1.01×	0.61×	0.71×	0.70×	71.72%	0.61×	0.91×	0.84×	91.92%	0.61×	0.91×	0.93×	91.92%
fdtd-2d	2.46×	0.56×	2.12×	0.63×	0.77×	0.70×	31.30%	0.63×	0.77×	1.15×	31.30%	0.77×	2.46×	1.37×	100%
fdtd-apml	7.98×	5.78×	1.00×	4.89×	7.36×	2.56×	92.23%	4.89×	7.36×	3.83×	92.23%	6.35×	7.36×	5.13×	92.23%
gauss-filter	1.83×	1.75×	1.00×	0.57×	1.13×	0.69×	61.75%	0.57×	1.13×	0.94×	61.75%	1.03×	1.13×	1.18×	61.75%
gemm	13.7×	2.74×	1.05×	2.94×	1.49×	1.54×	21.43%	2.94×	2.63×	2.59×	21.43%	2.94×	8.49×	5.43×	61.97%
gemver	1.95×	1.84×	1.44×	0.97×	0.97×	0.69×	49.74%	0.97×	0.97×	0.85×	49.74%	0.97×	1.95×	1.41×	100%
gesummv	2.44×	0.91×	2.42×	1.71×	1.94×	1.34×	79.51%	1.71×	1.94×	1.72×	79.51%	1.94×	1.94×	2.05×	79.51%
gramschm	10.9×	3.86×	1.01×	3.40×	1.00×	2.96×	31.05%	3.94×	1.00×	3.98×	36.30%	3.94×	1.01×	6.61×	36.30%
lu	1.67×	1.16×	1.01×	1.63×	1.15×	0.79×	97.60%	1.63×	1.15×	1.12×	97.60%	1.63×	1.53×	1.40×	97.60%
ludcmp	1.03×	0.96×	1.01×	1.01×	1.01×	1.02×	99.03%	1.01×	1.01×	1.02×	99.03%	1.02×	1.01×	1.02×	99.03%
mvt	1.48×	1.17×	1.00×	0.78×	1.03×	0.53×	69.59%	0.78×	1.03×	0.79×	69.59%	0.83×	1.03×	0.98×	69.59%
reg_detect	1.07×	0.52×	1.00×	0.29×	0.29×	0.67×	27.10%	0.29×	0.29×	0.82×	27.10%	0.62×	0.45×	0.97×	57.94%
seidel	9.71×	0.81×	1.00×	0.83×	0.83×	3.39×	8.55%	0.83×	0.98×	4.05×	8.55%	0.98×	7.60×	6.96×	78.27%
symm	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	100%	1.00×	1.00×	1.00×	100%	1.00×	1.00×	1.00×	100%
syr2k	7.57×	0.25×	7.15×	5.87×	7.14×	2.65×	94.32%	5.87×	7.14×	4.51×	94.32%	5.87×	7.14×	6.63×	94.32%
syrk	9.17×	0.78×	8.84×	3.76×	1.38×	1.99×	41.00%	3.76×	2.52×	2.90×	41.00%	4.62×	9.01×	5.23×	98.26%
trisolv	3.90×	1.40×	1.50×	3.69×	3.69×	1.07×	94.62%	3.69×	3.69×	1.48×	94.62%	3.69×	3.69×	2.18×	94.62%
trmm	1.27×	0.33×	1.00×	0.14×	0.14×	0.43×	11.02%	0.14×	0.14×	0.87×	11.02%	0.14×	1.20×	1.04×	94.49%
Average	6.1×	2.13×	1.98×	3.16×	3.27×	1.61×	64.39%	3.16×	3.43×	2.24×	65.38%	3.50×	4.68×	3.32×	83.18%

the compiler side. Our technique is able to compensate for this effect, by using simpler (in terms of code structure) polyhedral optimizations when it is the most profitable.

The 1-shot model can be seen as a non-iterative compilation scheme: the unseen program is analyzed once to gather its hardware performance counter values, and the model outputs the optimization to be applied. This model provides satisfactory improvements in the majority of cases, however for about 1/3 of the benchmarks applying the sequence predicted best will decrease the performance. We believe this model can be improved. We conducted additional experiments that includes tuning the learning algorithm parameters (e.g., the Gaussian parameter γ and the soft margin parameter C for the SVM), with improvements observed, but there was no single configuration that was performing best on all two machines. At this stage, tuning the learning algorithm parameters specifically for each machine remains an alternative. Clustering the benchmarks may also significantly simplify the learning problem, preliminary experiments indicate this is a promising direction.

The 2-shot model provides only a small improvement over the 1-shot, in contrast the 5-shot model can reach close to 2× better performance than the 1-shot. On average, SVM performs better than LR and Random on all machines when considering the 5-shot model. The 5-shot SVM model reaches on average 85%-89% of the space optimal performance improvement. This emphasizes the relevance of allowing for a limited empirical search step, in order to significantly improve the final performance gain.

We also observe that a pure random search on average let us discover significant performance improvements, however almost systematically lower than using LR or SVM. Furthermore, because of the uneven distribution of good points in the space, Random may fail to draw a good transformation sequence while the SVM and LR procedures are deterministic.

C. *Accuracy of the Prediction*

The model we build for LR and SVM predicts the speedup of a specific polyhedral optimization choice, given the performance counters of the program on which it would be applied. To estimate the accuracy of the prediction, we show in Figure 3-4 the performance predicted by LR and SVM for all candidate optimizations, sorted w.r.t. the actual performance of the optimizations, for four representative benchmarks.

Figure 3 compares the prediction for the same benchmark 2mm on both tested architectures. First, we observe the relatively low density of best points, represented at the far right of the plain curve Actual. This emphasizes the search problem is not trivial, and plain random techniques have a low probability in average to discover the optimal points. Regarding the prediction, we observe in both cases for SVM numerous spikes of predicted best points. A careful observation shows a slight difference in the speedup predicted for all spikes, leading to the highest spike for E5620 to correspond to one of the optimal best point; while for E7450 the 4 highest spikes do not achieve more than 2.38× improvement. This pattern is representative of several benchmarks: the models predicts a fraction of the search space to be potentially optimal, represented by those spikes. We have observed that in most cases a nearly optimal point is in the five first, however there are cases such as gauss-filter for which only the 9^{th} spike achieves the optimal speedup,

TABLE II

PERFORMANCE IMPROVEMENTS FOR INTEL XEON E7450 (BASELINE: ICC 11.0 -FAST)

				1-shot				2-shot				5-shot			
Benchmark	Opt	Poly	ICC	LR	SVM	R	%Opt	LR	SVM	R	%Opt	LR	SVM	R	%Opt
2mm	13.1×	3.67×	1.00×	1.67×	2.38×	2.77×	21.00%	1.67×	2.38×	5.13×	38.89%	12.96×	12.54×	9.32×	98.26%
3mm	12.1×	2.17×	1.00×	2.17×	1.32×	1.36×	17.89%	2.17×	11.7×	2.93×	96.62%	2.88×	11.7×	5.37×	96.62%
adi	3.28×	2.65×	1.33×	2.37×	0.42×	1.37×	72.26%	2.37×	0.42×	2.12×	72.26%	2.37×	0.46×	2.87×	72.26%
atax	1.96×	1.80×	1.00×	1.20×	0.22×	0.67×	61.22%	1.20×	0.22×	1.14×	61.22%	1.20×	0.22×	1.54×	61.22%
bicg	1.66×	1.01×	1.00×	1.54×	1.06×	0.84×	92.77%	1.54×	1.06×	1.08×	92.77%	1.66×	1.06×	1.40×	100%
cholesky	1.00×	0.98×	1.16×	0.61×	0.98×	0.76×	98.99%	0.61×	0.98×	0.79×	98.99%	0.62×	0.98×	0.93×	98.99%
correlation	36.6×	36.6×	12.3×	36.6×	20.2×	12.1×	100%	36.6×	33.6×	13.3×	100%	36.6×	33.6×	22.2×	100%
covariance	36.9×	11.0×	9.87×	24.5×	24.5×	8.50×	66.30%	24.5×	24.5×	16.5×	66.30%	36.2×	34.2×	25.7×	98.18%
doitgen	18.3×	5.21×	1.00×	6.91×	1.61×	2.37×	37.78%	6.91×	5.64×	2.94×	37.78%	6.91×	5.64×	5.35×	37.78%
durbin	1.00×	1.00×	1.00×	0.99×	1.00×	1.00×	100%	0.99×	1.00×	1.00×	100%	1.00×	1.00×	1.00×	100%
dynprog	1.00×	0.41×	1.00×	0.64×	0.96×	0.75×	96.97%	0.64×	0.96×	0.84×	96.97%	0.64×	0.96×	0.92×	96.97%
fdtd-2d	3.06×	1.71×	2.95×	1.71×	1.71×	1.16×	55.88%	1.71×	2.06×	1.43×	67.32%	1.71×	3.06×	1.94×	100%
fdtd-apml	6.25×	3.01×	1.00×	3.85×	3.30×	1.88×	61.60%	3.85×	4.41×	3.02×	70.56%	3.85×	4.41×	4.02×	70.56%
gauss-filter	1.06×	0.94×	1.00×	0.35×	0.35×	0.55×	33.01%	0.35×	0.35×	0.65×	33.01%	0.35×	0.35×	0.71×	33.01%
gemm	11.6×	3.90×	1.00×	3.43×	2.78×	1.39×	29.54%	3.43×	2.78×	3.68×	31.70%	3.43×	11.0×	6.27×	95.00%
gemver	2.68×	2.29×	1.14×	2.18×	2.59×	1.08×	96.64%	2.18×	2.59×	1.81×	96.64%	2.67×	2.64×	2.26×	99.63%
gesummv	1.45×	0.68×	1.44×	1.24×	0.84×	1.02×	85.52%	1.24×	0.84×	1.21×	85.52%	1.24×	0.92×	1.27×	85.52%
gramsch	4.34×	2.91×	2.61×	0.83×	0.83×	1.60×	19.12%	0.83×	0.83×	1.84×	19.12%	1.09×	1.09×	2.74×	25.11%
lu	7.24×	3.15×	1.15×	0.43×	0.43×	1.70×	5.93%	0.43×	0.96×	2.63×	13.25%	0.96×	1.84×	4.19×	25.41%
ludcmp	1.00×	0.99×	1.00×	0.99×	0.99×	0.99×	99.00%	0.99×	0.99×	1.00×	99.00%	0.99×	1.00×	1.00×	100%
mvt	1.75×	1.70×	1.00×	1.73×	1.73×	0.97×	98.86%	1.73×	1.73×	1.02×	98.86%	1.73×	1.73×	1.63×	98.86%
reg_detect	1.11×	0.80×	1.05×	0.12×	0.05×	0.42×	37.84%	0.31×	1.02×	0.73×	91.89%	0.31×	1.06×	1.02×	95.50%
seidel	9.92×	1.54×	1.00×	9.45×	9.45×	2.52×	95.26%	9.45×	9.45×	3.62×	95.26%	9.45×	9.45×	5.84×	95.26%
symm	1.02×	1.00×	1.02×	0.83×	0.83×	0.93×	81.37%	0.83×	0.83×	0.95×	81.37%	0.83×	0.83×	1.00×	81.37%
syr2k	22.7×	22.7×	22.7×	22.7×	22.7×	6.91×	100%	22.7×	22.7×	8.87×	100%	22.7×	22.7×	18.4×	100%
syrk	19.7×	9.10×	19.6×	7.85×	2.14×	2.94×	39.77%	7.85×	3.18×	7.04×	39.77%	7.85×	19.6×	8.58×	99.09%
trisolv	1.97×	0.98×	1.00×	1.26×	1.26×	0.85×	63.85%	1.26×	1.26×	1.24×	62.94%	1.26×	1.42×	1.38×	72.08%
trmm	1.16×	0.65×	1.03×	0.40×	0.04×	0.57×	34.48%	0.40×	0.04×	0.84×	34.48%	0.40×	0.99×	1.00×	86.11%
Average	8.04×	4.48×	3.38×	4.91×	3.77×	2.14×	64.42%	4.92×	4.91×	3.19×	70.80%	5.82×	6.60×	4.99×	82.95%

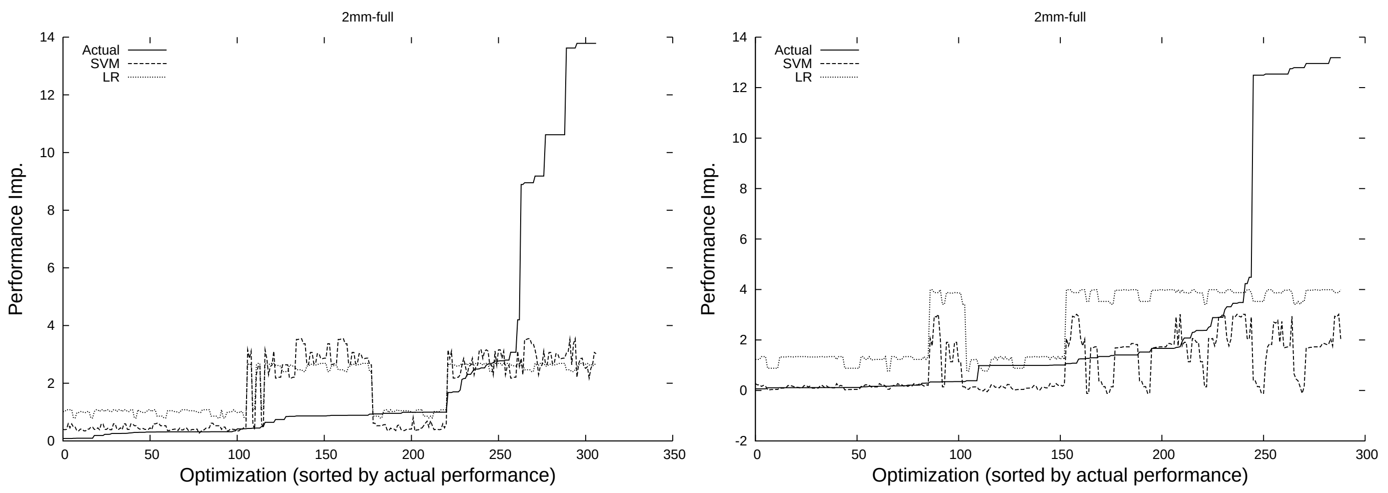

Fig. 3. 2mm Prediction for Xeon E5620 (left) and Xeon E7450 (right)

as shown in Figure 4.

In general, LR prediction follows the prediction of SVM, but with a smoother behavior. This is particularly shown in Figure 5. For such situation, LR simply fails to differentiate enough between all the variants in the search space, and is unable to isolate the best sequences: the obtained speedup tops at 11.8× with LR, while it reaches 17.84× with SVM.

We also confirmed the need to create models for each architecture to be considered. Different shapes of the performance distribution indicate that the quantity of good performing transformation sequences vary from one architecture to the other. In addition, the back-end compiler is part of the problem and may trigger different optimization heuristics for different architectures, and we observed the compiler optimization flow is extremely hard to predict and can easily be disturbed by a high-level transformation. Learning a model for each architecture is a valid alternative to circumvent those issues.

Discussions: We also investigated using a 10-shot SVM model, which takes the ten predicted best sequences and evaluate all of them. This 10-shot model improves the performance by reaching an average 95% of the search space optimal performance improvement. It also helped improving

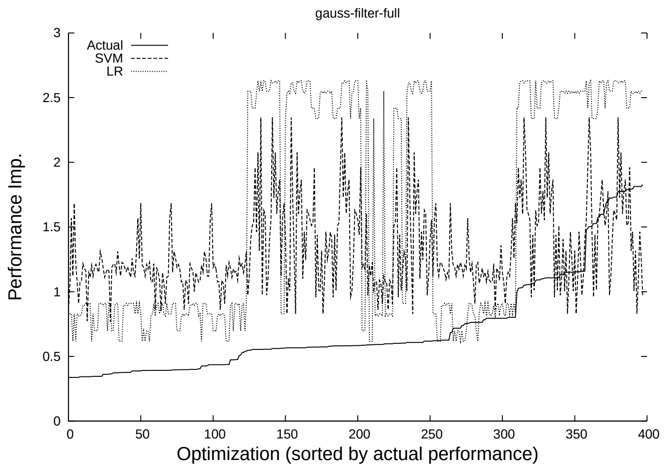

Fig. 4. Gauss-Filter Prediction for Xeon E5620

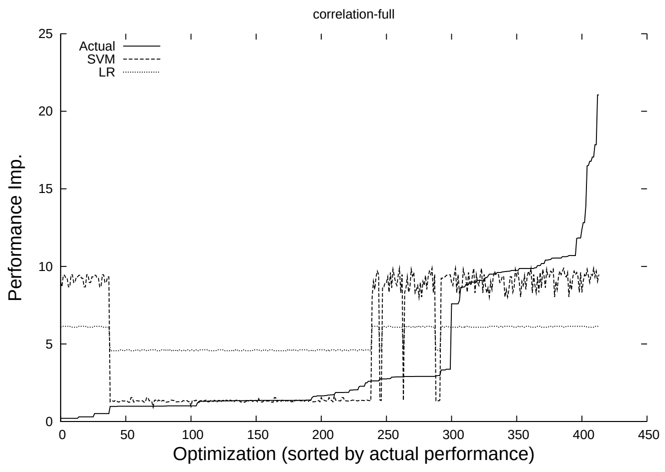

Fig. 5. Correlation Prediction for Xeon E5620

the performance of the more problematic benchmarks such as doitgen or gemm and reached the space optimal performance for those.

As a future work, we will also investigate clustering the benchmarks into several categories to simplify the learning process and improve the overall prediction quality on each cluster. However preliminary results indicates an efficient clustering corresponds to isolating the benchmarks on which tiling in our framework prevents ICC from performing the same optimizations as without it, thereby emphasizing the sensibility of the clustering to the back-end compiler features. If such pattern is confirmed, it opens the research problem of how to *characterize* the compiler optimization features, and to integrate the result into the performance models.

V. Related Work

In recent years, considerable research has addressed iterative compilation and its benefits have been reported in several publications [22], [10], [11], [15], [19], [1]. Iterative compilation has been shown to regularly outperform the most aggressive compilation settings of commercial compilers, and it has often been comparable to hand-optimized library functions [39], [16], [33], [38].

Deciding the enabling or disabling of loop unrolling was done by Monsifrot *et al.* [26] using decision tree learning, and was one of the early efforts on using machine learning to tune a high-level transformation. Kulkarni et al. [23] introduced a system that used databases to store previously tested code, thereby reducing running time. They also disabled some optimizations that did not seem to improve the running time of the kernel. These techniques are very expensive and therefore only effective when programs are extremely small, such as those used in embedded domains. Cooper et al. [10] used genetic algorithms to address the compilation phase-ordering problem. They were concerned with finding “good” compiler optimization sequences that reduced code size. Their technique was successful in reducing code size by as much as 40%. However, their technique is application-specific — a genetic algorithm had to *retrain* for each program to decide the best optimization sequence for that program.

An innovative approach to iterative compilation was proposed by Parello et al. [29] where they used performance counters at each stage to propose new optimization sequences. The proposed sequences were evaluated and the measured performance counters with them were used to choose new optimizations to try. Even though this was a very systematic approach, the time required for this method was almost several weeks for each benchmark. Our technique does not need to generate performance counters during each iteration, but instead produces a single model to predict the best optimization sequences for a program.

Cavazos et al. address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics [8]. That work was limited to the traditional optimization space of the PathScale compiler. Despite the numerous transformations considered, the complexity is not cmparable to the restructuring transformations automatically generated by the polyhedral framework.

Chen *et al.* developed the CHiLL infrastructure [9], a polyhedral loop transformation and code generation framework. Tiwari *et al.* [35] coupled the Active Harmony search engine to automatically tune some high-level transformation parameters, such as tile sizes. In this paper we target quite a different search space, going tuning the individual parameters of a transformation: we balance the trade-off between several possibly contradictory objectives, such as parallelization, data locality enhancement and vectorization, demonstrating our results on a variety of benchmarks and machines.

Pouchet et al. performed empirical search to directly find the coefficients of the affine scheduling matrix in a polyhedral framework. [31]. While the results showed significant improvements on small kernels, the empirical search needed up to a thousand runs for larger benchmarks [30]. In this work, we have abstracted the scheduling matrix behind high-level polyhedral primitives and the associated cost models for selecting the enabling transformations, reducing the search space to only a few hundred possibilities in place of the billions of possible schedules. This enabled us achieve on average 85% of the search-space-optimal performance in no more than 5 runs.

VI. Conclusion

The problem of improving performance through compiler optimization has been extensively studied, in particular to improve the portability of the optimization process across a variety of architectures. Iterative compilation and machine learning techniques have been demonstrated as powerful mechanisms to automatically compute good compiler flags, improving the speed of the generated program and automatically adapting to the target architecture.

However, in the multi-core era with increasingly complex hardware, very advanced high-level transformation mechanisms are required to efficiently map the program on the target machine. Complex sequences of loop transformations are needed to implement tiling, parallelization and vectorization all together. While all these optimizations have been studied independently, in practice they must be combined to optimize performance.

A modern loop nest optimizer faces the challenge of sometimes contradictory cost models, simply because there is no single solution that may maximize parallelism, vectorization, data locality and still achieve the best performance. Very little work has been done to date in using learning models for selecting high-level transformations, to drive a loop nest optimizer that operates on a very rich and complex search space. Our work is the first to propose the use of learning models to compute effective loop transformations in the *polyhedral model*, encompassing tiling, parallelization, vectorization and data locality improvement via high-level primitives. To determine the best loop transformations for a program, we decompose the problem into (1) searching for the best sequence of high-level polyhedral primitives (e.g., tiling, vectorization, etc.); and (2) using static cost models to compute the final sequence of elementary loop transformations that implement those primitives.

In this work, we leverage the power of the polyhedral transformation framework to automatically build very complex sequences of transformations, enabling tiling and parallelization transformations on a wide range of numerical codes. To select an effective optimization in this space, we have implemented a speedup predictor model that correlates the runtime characteristics of a program (modeled with performance counters) with the speedup expected from a given polyhedral optimization (modeled with a sequence of high-level primitives). We evaluated our approach using two machine learning algorithms, linear regression and support vector machine, on a variety of benchmarks and two multi-core machines. For the test suite, the best points in our optimization search space yield an average 8× speedup (with peaks of up to 36×) over ICC on an Intel Xeon E7450. Using the predictive machine learning models, testing at most five candidate optimizations on the target machine, we achieve an average speedup of 6.6× over the Intel ICC compiler, which corresponds to an average of 83% of the best possible performance among all points in the entire search space,

Acknowledgments*:* This work was funded in part by the U.S. National Science Foundation through award 0926688, the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, the DARPA Computer Science Study Group (CSSG) and NSF Career award 0953667.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *4th Annual International Symposium on Code Generation and Optimization (CGO)*, Mar. 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, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Intl. Conf. on Compiler Construction (ETAPS CC'10)*, LNCS 6011, pages 283–303, Paphos, Cyprus, Mar. 2010.

[4] 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.

[5] 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.

[6] R. R. Bouckaert, E. Frank, M. A. Hall, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. WEKA–experiences with a java open-source project. *Journal of Machine Learning Research*, 11:2533–2541, 2010.

[7] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, October 2006.

[8] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, pages 185–197, 2007.

[9] 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.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, Georgia, July 1999. ACM Press.

[11] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, August 2002.

[12] C. Dubach, J. Cavazos, B. Franke, M. O'Boyle, G. Fursin, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the ACM International Conference on Computing Frontiers*, May 2007.

[13] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.

[14] 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.

[15] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 78–86, New York, NY, USA, 2005. ACM Press.

[16] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on ”Program Generation, Optimization, and Platform Adaptation”.

[17] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, P. Barnard, E. Ashton, E. Courtois, F. Bodin, E. Bonilla, J. Thomson, H. Leather, C. Williams, and M. O'Boyle. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[19] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[20] INRIA and The Ohio State University. Polybench, the polyhedral benchmark suite. http://www-rocq.inria.fr/~pouchet/software/polybench.

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

[22] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[23] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 171–182, New York, NY, USA, 2004. ACM Press.

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

[25] 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.

[26] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proc. of the 10th Intl. Conf. on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.

[27] P. Mucci. Papi – the performance application programming interface. http://icl.cs.utk.edu/papi/index.html, 2000.

[28] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *Intl. Conf. on Compilers Architectures and Synthesis for Embedded Systems (CASES'10)*, Oct. 2010.

[29] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[30] 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.

[31] 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.

[32] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *ACM Supercomputing Conf. (SC'10)*, New Orleans, Lousiana, Nov. 2010. 11 pages.

[33] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. special issue on ”Program Generation, Optimization, and Platform Adaptation”.

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

[35] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[36] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, North Carolina, Sept. 2009.

[37] 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)*, pages 102–113, Seattle, Washington, Mar. 2009.

[38] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, 2004.

[39] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[40] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 2000.

[41] M. Wolfe. More iteration space tiling. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 655–664, New York, NY, USA, 1989. ACM.

[42] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*, San Diego, CA, June 2003.