Predictive Modeling in a Polyhedral Optimization Space

Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, P. Sadayappan

Noname manuscript No.
(will be inserted by the editor)

Predictive Modeling in a Polyhedral Optimization Space

Eunjung Park[1] · **John Cavazos**[1] ·
Louis-Noël Pouchet[2,3] · **Cédric Bastoul**[4] ·
Albert Cohen[5] · **P. Sadayappan**[2]

Received: April 2012 / Accepted: September 2012

Abstract High-level program optimizations, such as loop transformations, are critical for high performance on multi-core targets. However, complex sequences of loop transformations are often required to expose parallelism (both coarse-grain and fine-grain) and improve data locality. The polyhedral compilation framework has proved to be very effective at representing these complex sequences and restructuring compute-intensive applications, seamlessly handling perfectly and imperfectly nested loops. It models arbitrarily complex sequences of loop transformations in a unified mathematical framework, dramatically increasing the expressiveness (and expected effectiveness) of the loop optimization stage. Nevertheless identifying the most effective loop transformations remains a major challenge: current state-of-the-art heuristics in polyhedral frameworks simply fail to expose good performance over a wide range of numerical applications. Their lack of effectiveness is mainly due to simplistic performance models that do not reflect the complexity today's processors (CPU, cache behavior, etc.).

We address the problem of selecting the best polyhedral optimizations with dedicated machine learning models, trained specifically on the target machine. We show that these models can quickly select high-performance optimizations with very limited iterative search. We decouple the problem of selecting good complex sequences of optimizations in two stages: (1) we narrow the set of candidate optimizations using static cost models to select the loop transformations that implement specific high-level optimizations (e.g., tiling, parallelism, etc.); (2) we predict the performance of each high-level complex optimization sequence with trained models that take as input a performance-counter characterization of the original program.

Our end-to-end framework is validated using numerous benchmarks on two modern multi-core platforms. We investigate a variety of different machine learning algo-

This article is an extended version of our work published at CGO'11 [43].

[1] University of Delaware, USA. {epark,cavazos}@cis.udel.edu · [2] The Ohio State University, USA. {pouchet, saday}@cse.ohio-state.edu · [3] University of California Los Angeles, USA. pouchet@cs.ucla.edu · [4] LRI, University of Paris-Sud 11, France. cedric.bastoul@u-psud.fr · [5] INRIA Paris-Rocquencourt / ENS, France. albert.cohen@inria.fr

rithms and hardware counters, and we obtain performance improvements over productions compilers ranging on average from 3.2× to 8.7×, by running not more than 6 program variants from a polyhedral optimization space.

Keywords Loop transformation · polyhedral optimization · iterative compilation · · machine learning · performance counters

1 Introduction

Numerous scientific and engineering compute-intensive applications spend most of their execution time in loop nests that are amenable to high-level optimizations. Typical examples include dense linear algebra codes (e.g. [4,6,57]) and stencil-based iterative methods (e.g. [19, 40, 50]). Those applications are typically executed on multicore architectures, where the data access cost is hidden behind complex memory hierarchies. High-level loop transformations are critical to achieving high performance in such context to correctly exploit the various levels of parallelism (course-grained versus fine-grained) available and to leverage the program's data locality potential. However, the best loop optimization sequence is program-specific and depends on the features of the target hardware. Thus, tuning the high-level loop transformations is critical to reach the best possible performance as illustrated by Pouchet *et al.* [44–46].

Although significant advances have been made in developing robust and expressive compiler optimization frameworks, identifying the best high-level loop transformations for a given program and architecture remains an open problem. Manually-constructed heuristics are used to identify good transformations, but they rely on overly simplified models of the machine. These simple static models are unable to characterize the complex interplay between all the hardware resources (e.g., cache, TLBs, instruction pipelines, hardware prefetch units, SIMD units, etc.). Moreover, optimization strategies often have conflicting objectives: for instance maximizing thread-level parallelism may hamper SIMD-level parallelism and can degrade data locality.

In the quest for performance portability, the compiler community has explored research based on iterative compilation and machine learning to *tune the compiler optimization flags or optimization passes* to find the best set of optimizations for a given combination of benchmarks and target architectures. Although significant performance improvements have been demonstrated [1, 27, 37, 39], the performance obtained has generally been limited by the optimizations selected for automatic tuning and by the degrees of freedom available for exploration.

The *polyhedral optimization framework* has been demonstrated as a powerful alternative to traditional compilation frameworks. Polyhedral frameworks can optimize a restricted, but important, set of loop nests that contain only affine array accesses. For loops that are amenable to polyhedral compilation, these frameworks can model an arbitrarily complex sequence of loop transformations in a single optimization step within a powerful and unified mathematical framework [22, 23, 31, 32, 35, 59]. The downside of this expressiveness is the difficulty in selecting an *effective* set of affine transformation coefficients that result in the best combination of tiling, coarse- and

fine-grain parallelization, fusion, distribution, interchange, skewing, permutation and shifting [13, 28, 46, 53].

Past and current work in polyhedral compilation has contributed algorithms and tools to expose model-driven approaches for various high-level transformations, including (1) loop fusion and distribution to partition the program into independent loop nests, (2) loop tiling to partition (a sequence of) loop nests into blocks of computations, (3) thread-level parallelism extraction, and (4) SIMD-level parallelism extraction. There has been some recent limited success at developing analytical cost models to select good complex optimization sequences in the polyhedral model. For example, Bondhugula et al. proposed the first integrated heuristic for parallelization, fusion, and tiling in the polyhedral model [8, 9] 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. Nevertheless, it has been shown that these simple static models are ineffective at systematically select the most effective transformation on a range of numerical applications [46]. Previous work on iterative compilation based on the polyhedral framework showed that there are opportunities for large performance improvements over native compilers [3, 44–46, 53], significantly outperforming compilation flag tuning, optimization pass selection, or optimization phase-ordering. However, directly tuning the polyhedral transformation in its original abstract representation remains a highly complex problem because the search space is usually infinite. Despite progress in understanding the structure of this space and how to bound its size [47], this problem remains largely intractable in its original form.

We now summarize the contributions of the current article. We address the problem of effectively balancing the trade-off between data locality and various levels of parallelism in a large set of 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 solid performance improvements by using the high-level transformation that was predicted best by our model, improving performance on average by $2\times$ up to $7\times$ over the native compiler. To the best of our knowledge, this is the first effort that demonstrates very effective discovery of complex high-level loop transformations within the polyhedral model using machine learning models. A performance that is close to the search-space-optimal can be attained by evaluating no more than 6 optimized program versions, using an iterative compilation approach. We explore different learning algorithms for building our models and report their ability to predict good polyhedral transformations. We observe that while no single algorithm is systematically the best

for all benchmarks, by combining models we can reach 8.7× average performance improvement over the native compiler, by testing no more than 6 program versions. Finally, we study feature reduction on our set of performance counters and show that only a handful of counters is required to characterize programs for this problem.

In Section 2, we first present details on the optimization space we consider before analyzing the performance distribution of the considered search space in Section 3. We present our machine learning approach to select good optimizations in Section 4. Experimental results are presented in Section 5. We discuss related work in Section 6.

2 Optimization Space

We now present the optimization search space we consider in this work. Any candidate optimization in this space can be automatically computed and applied on the input program, thus producing a transformed variant to be executed on the target machine. Deciding how to select an optimization in this space is the subject of the later Section 4.

2.1 Overview of the Approach

High-level loop transformations are crucial to effectively map a piece of code onto a particular processor. 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 way data is accessed by the code may need to be reorganized to better exploit the cache memory hierarchy and improve communication costs. 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 [9, 23, 31, 35, 49, 54].

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 [28, 36] or syntactic-based [12], was the challenge of computing the enabling sequence that was required to apply the main optimization primitives. This led most previous work to be limited in applicability: the enabling transformations were not considered in a separate fashion, so that transformations such as tiling and coarse-grained parallelization could not be applied in the most effective fashion.

We address this issue by decoupling the problem of selecting a polyhedral optimization into two steps: (1) select a sequence of high-level optimizations, from the set shown in Table 1, this selection being based on machine learning and feedback from hardware performance counters; and (2) for the selected high-level optimizations, use *static cost models* to compute the appropriate enabling transformations. We thus

keep the expressiveness and applicability of the polyhedral model, while focusing the selection decision only on the main transformations.

Table 1 shows the various high-level optimizations we consider and their parameter range, they are each described in the following sections. For each of these high-level optimizations, we rely on a polyhedral-based transformation framework to compute any required *enabling* loop transformation. If such sequence of enabling loop transformations exist, it will be found by the static model.

Table 1 High-level primitives considered in this work

High-level optimization	Possible values
loop fusion / distribution	max-fuse, smart-fuse, no-fuse
loop tiling	tile size (one per tiled loop) :1 (no tiling), 32
wavefronting	on, off
thread-level parallelization	on, off
pre-vectorization	on, off
SIMD-level parallelization	on, off
register-tiling	unroll factors: 1 (no unrolling), 2, 4, 8

We remark that high-level transformations are by far not sufficient to achieve optimal performance. Numerous low-level optimizations are also required, and chip makers such as Intel have developed extremely effective closed-source compilers for their processors.

We consider such compilers as black-boxes, because of the difficulty in precisely determining which optimizations are implemented and when. Technically, the loop optimization stages of those compilers may interact with our source-level transformations in a detrimental way, by either altering the loop structure we generated, and/or by becoming unable to apply a profitable loop optimization on the code we have generated. A typical example is SIMD vectorization, which may or may not be successfully applied by the back-end compiler on the transformed program we generated, even if the resulting program is indeed vectorizable. Consequently, our optimization scheme may result in sub-optimal performance if for instance the compiler is unable to apply SIMD vectorization on our transformed program while it was able to apply it on the original program or some other variants. A precise and fine-grain tracking of the back-end compiler optimizations applied would be required to avoid this potential issue, but we have not addressed this problem in the present work. We also highlight in Section 3 that indeed the optimal high-level transformation we generate must be compiler specific. 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.

2.2 Polyhedral Model

Sequences of (possibly imperfectly nested) loops amenable to polyhedral optimization are called *static control parts* (SCoP) [23,28] 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 [7], 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 loop transformations [28]. 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 [23], 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 [28].

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 [5] to perform this task.

We used the open-source polyhedral compiler PoCC[1] for this paper, and we have extended it for the purposes of enabling more effective model-based program transformations.

2.3 Loop Tiling

Tiling is a crucial loop transformation for parallelism and locality. It partitions the computation into rectangular blocks that can be executed atomically. When tiling is applied on a program, we rely on the Tiling Hyperplane method [9] 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. First, 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 a tile should typically reside in the L1 cache. The problem of selecting the optimal tile size is known to be difficult and empirical search is often used for

[1] http://pocc.sourceforge.net

high-performance codes [55, 58, 60]. 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.

2.4 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 this seminal formulation, Bondhugula proposed to find a transformation that maximizes the number of fused statements on the whole program using an Integer Linear Program (ILP) encoding of the problem [8]. 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. Pouchet et al. showed that iterative search among the possible fusion structures can provide significant performance improvement [46, 47]. However, to control the size of the search space we rely on three specific fusion / distribution schemes that proved to be effective for a wide variety of programs. The three high-level fusion schemes we consider in this paper are: (1) no-fuse, where we do not fuse at all; (2) smart-fuse, where we only fuse together statements that carry data reuse and of similar loop nesting depth; and (3) max-fuse, where we 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.

2.5 Wavefronting

When a loop nest is tiled, it is always possible to execute the tiles either in parallel or in a pipeline-parallel fashion. *Wavefronting* is the transformation creating a valid, pipeline-parallel schedule for the tiled execution. It is useful only to expose coarse-grain parallelism between tiles, when the outer-most tile loop is not already sync-free parallel. When wavefronting is turned on, the tile loops are modified to expose parallelism at the expense of increasing the distance between reused elements. So, there is a trade-off to the application of this transformation, and as such is part of our high-level optimization choices. Additionally, we remark that wavefronting will not be useful for a program where thread-level parallelism is not also useful.

2.5.1 Pre-vectorization

The tiling hyperplane method finds a loop transformation that pushes dependences to the inner loop levels, naturally exposing coarse-grain parallelism. Nevertheless, for effective SIMD execution it is desirable to expose at least one level of *inner parallelism*. The pre-vectorization stage modifies the affine schedule of the program (that is, the transformation to be applied) so that the inner-most parallel loop dimension is sunk into the inner-most loop dimension.

This approach can guarantee that the inner-most loop(s) of a program are sync-free parallel, when the dependence permits. It has the advantage of enforcing the parallelism of *all loops at a given depth* in the generated loop nest, for the entire program. However this simple model shows significant limitations: no information is taken into account regarding the contiguity of data accesses, a critical concern for effective SIMD execution. Also, it works as a pre-pass before the code is transformed, i.e., it does not take into account the changes in the loop structure that is going to be generated by CLooG (our polyhedral code generator). Nevertheless, this model has shown potential to increase the success of the new SIMD-level parallelization pass that we have implemented in PoCC (detailed next).

2.5.2 SIMD-level parallelization

Our approach to vectorization extends recent analytical modeling results by Trifunovic et al. [54]. We take advantage of the polyhedral representation to restructure imperfectly nested programs allowing us to expose vectorization opportunities in 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 [54]. This is a more SIMD-friendly approach than simple pre-vectorization. We note however that this interchange may not always lead to the optimal vectorization because of the limitations of the model or may simply be useless for a machine which does not support SIMD instructions. In addition, we have implemented this vectorization transformation pass on the generated code after applying the tiling hyperplane method. Our algorithm operates on individual loop nests generated after the separation phase of CLooG and

does not require to have all loops at a given depth in the program to be vectorized in a similar fashion. To achieve this, we benefit from the fact that programs generated by CLooG are also polyhedral programs, that is we can re-analyze the transformed code and extract a polyhedral representation from it. When SIMD vectorization is turned on, we mark the vectorizable loops with `ivdep` and `vector always` pragmas to facilitate the task of the compiler auto-vectorization.

2.5.3 Thread-level parallelization

Thread-level parallelism is not always beneficial, e.g., with small kernels that execute few iterations or when it prevents vectorization. It is thus critical to be able to disable thread-level parallelism in some instances. We have designed a specific optimization pass in PoCC that analyzes the *generated code*, in a similar fashion to SIMD-level parallelization. It finds the outer-most parallel loop in each loop nest in the generated code using automated scalar privatization techniques. When this optimization is turned on, we use OpenMP to generate parallel code and insert a `#pragma omp parallel for` above the outer-most parallel loop that was found.

2.5.4 Register Tiling

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. Register tiling, or unroll-and-jam, combines the power of unrolling multiple permutable loops in the inner loop body. In our framework, register tiling is performed as a post-pass considering only the inner-most two loops in a loop nest, if they are permutable. We expose four different sizes for the unroll factor, which is the same for both loops to be unroll-and-jammed: 1 (no unrolling), 2, 4 and 8.

2.6 Putting it all Together

A sequence of high-level optimizations 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 (e.g., SIMD-level parallelization and thread-Level parallelization) are encoded using a single bit.

Non-binary optimizations 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 three binary variables (x,y,z). The pair $(0,0,0)$ denotes no unroll-and-jam, an unroll factor of 2 is denoted by $(0,0,1)$, an unroll factor of 4 is denoted by $(0,1,1)$, and unroll factor of 8 by $(1,1,1)$. 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 no-fuse, smart-fuse or max-fuse);
2. Apply the Tiling Hyperplane method [9] locally on each partition to obtain a schedule for the program that (a) implements the fusion choice, (b) maximizes the number of parallel loops, and (c) maximizes the number of tilable dimensions [8] on each individual partition;
3. Modify the schedule according to the pre-vectorization model, if pre-vector is set, to expose inner parallel loops;
4. Modify the schedule to generate a wavefront parallel schedule, if wavefronting is set.
5. Tile all tilable loop nests, if any, if tile is set. The tile sizes to be used are encoded in T.
6. Extract the polyhedral representation of the generated code, for subsequent analysis.
7. Apply the per loop nest SIMD-level parallelization pass, if set.
8. Apply the per loop nest thread-level parallelization pass, if set.
9. Perform register tiling, if register-tiling is set. The unroll factors to be used are encoded in T.

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 an optimization has no effect on the final program because of semantic considerations, multiple values of T lead to the same candidate code version. Nevertheless, the applicability of those optimizations directly derives from the expressiveness of the polyhedral model, which significantly improves over other existing frameworks.

The search space, considering only values of T leading to distinct transformed programs, ranges from 91 to 432 in our experiments, out of 1152 possible combinations that can be encoded in T.

3 Analysis of the Performance Distribution

We now present an extensive quantitative analysis of the performance distribution of the search space of polyhedral transformations that we have built. Section 3.1 presents the machines and benchmarks we experimented with. We then extensively discuss the performance distribution for numerous benchmarks, machines and compilers, providing experimental evidence of the specificity of the optimal transformation to each of these three aspects.

3.1 Experimental Setup

We provide experimental results on two multi-core systems: Nehalem, a 2-socket 4-core Intel Xeon 5620 (16 H/W threads) with 16 GB of memory, and Q9650, a single socket 4-core Intel Quad Q9650 with 8 GB of memory. The back-end compilers used for the baseline and all candidate polyhedral optimizations are Intel ICC with option

`-fast` and GCC with option `-O3`. ICC version 11.1 and GCC version 4.5 were used for Nehalem, and ICC version 10.1 and GCC version 4.5 were used for Q9650. Thus, we present results for four different machine-compiler configurations: (1) Nehalem-ICC11.1 (2) Nehalem-GCC4.5 (3) Q9650-ICC10.1 and (4) Q9650-GCC4.5.

Our benchmark suite is PolyBench v2.1 [30] composed of 30 different kernels and applications containing static control parts. The list of programs in PolyBench is shown in Table 2. We used the reference datasets [30].

Prog. Name	Description
2mm	2 Matrix Multiplications (D=A×B; E=C×D)
3mm	3 Matrix Multiplications (E=A×B; F=C×D; G=E×F)
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
dynprog	Dynamic programming (2D)
fdtd-2d	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gauss-filter	Gaussian Filter
gemm	Matrix-multiply C = αA×B + βC
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition
mvt	Matrix Vector Product and Transpose
reg-detect	2-D Image processing
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank−2k operations
syrk	Symmetric rank−k operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Table 2 The 30 programs in PolyBench V2.1 were used for our training and testing of each prediction model. They are set of programs including computations used in data mining, image processing, and linear algebra solvers and kernels

3.2 Overview of the Performance Distribution

We first propose to quantify the performance distribution, from the perspective of determining the fraction of the search space that achieves better performance than the original code, and the fraction that achieves a nearly optimal performance in the

considered space. Figure 1 plots, for all considered architectures and compilers, the relative speedup (compared to the original program) achieved by each variants, for the covariance benchmark. Figure 2 shows a similar plot for the fdtd-2d benchmark.

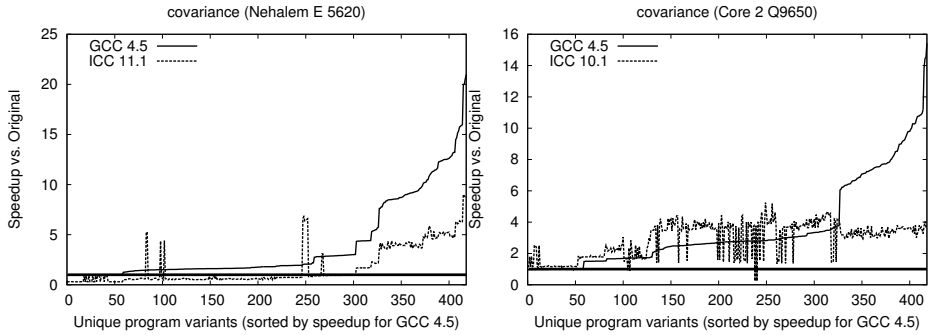

Fig. 1 Performance distribution for covariance

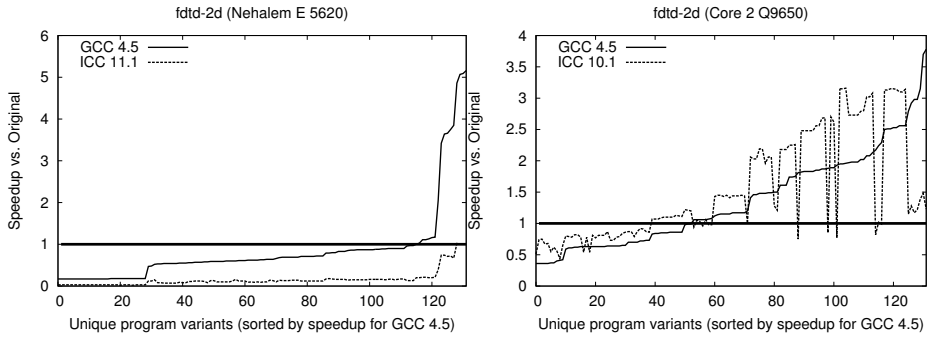

Fig. 2 Performance distribution for fdtd-2d

First, we observe that the fraction of the search space which improves performance is dependent upon the benchmark. covariance and fdtd-2d are two representative benchmarks. For covariance the majority of the search space improves performance, for both architectures and compilers. This means that a simple random approach is likely to succeed in improving performance for this benchmark. For fdtd-2d and the Nehalem architecture, the opposite pattern is observed where only a marginal fraction of the space improves performance. This latter benchmark corresponds to the majority of benchmarks, where usually a significant fraction of the space degrades performance. Indeed, this result confirms that the search space we consider is very expressive and that many of the high-level optimizations shown in Table 1 can actually *decrease* performance if not parameterized properly.

More importantly, we observe for all cases that the fraction of the space which achieves nearly optimal performance is extremely small, usually below 1% of the space. This pattern was observed for all benchmarks. This severely limits the ability of any naive statistical model to discover the best performance in this space, thus motivating our use of the powerful machine learning algorithms instead.

3.3 Variability Across Compilers

We now provide observations about the relative performance of similar transformations when used for the same machine and benchmark, but with two different compilers. Figure 3 and Figure 4 plots, for the two machines, the performance distribution for gemm and seidel using GCC and ICC, but sorted by increasing performance for GCC.

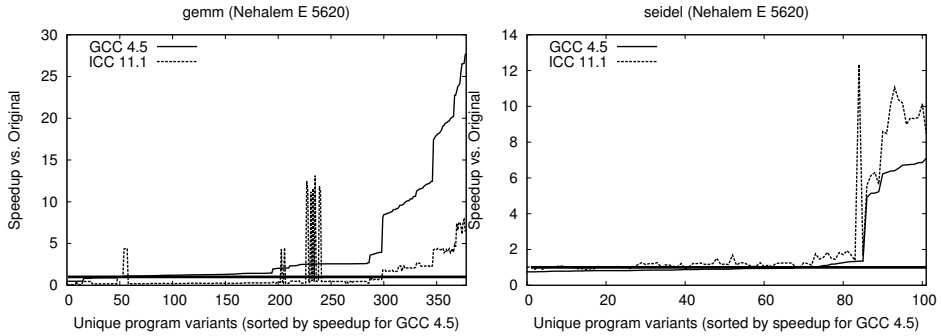

Fig. 3 Performance distribution for gemm and seidel for the E-5620 machine

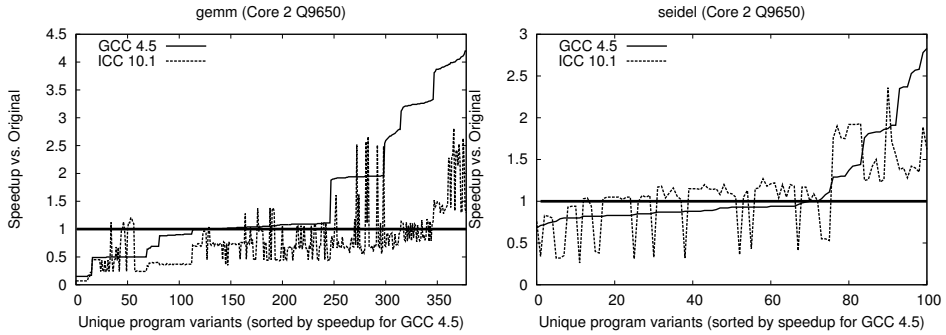

Fig. 4 Performance distribution for gemm and seidel for the Q-9650 machine

The most prominent feature of these plots is that the best performing optimized variants for GCC (on the far right of the plots) are not the best performing variants

for ICC, and conversely. That is, *the sequences of high-level optimizations achieving the best performance differ for each compiler*. This is particularly shown in Figure 3 for gemm or in Figure 4 for seidel. This pattern is not systematic for all benchmarks, but is dominant in our test suite. This can for instance be observed also in Figure 1 and Figure 2. One of the main reason for these differences comes from the very fragile optimization heuristics implemented in each compiler. The benefit of using our tool-chain to generate potentially effective transformations (e.g., tiling or register tiling) can break the application of high-performance scalar optimizations by the native compiler. This is because the transformed code to be compiled becomes syntactically more complex, and fragile optimization cost models in the native compiler can be challenged by its structure. Another reason comes naturally from the differences in optimizations being implemented in each compiler and when they are applied. For instance, we have manually observed instances where the cost model of GCC applies register tiling more aggressively than ICC, thus making useless (if not detrimental) the application of register tiling in our framework.

3.4 Variability Across Machines

We now present experimental evidence of the sensitivity of transformation sequences to the target machine. In Figure 5 we plot, for the same benchmark and the same compiler, the performance distribution for both machines sorted by ascending performance for Nehalem.

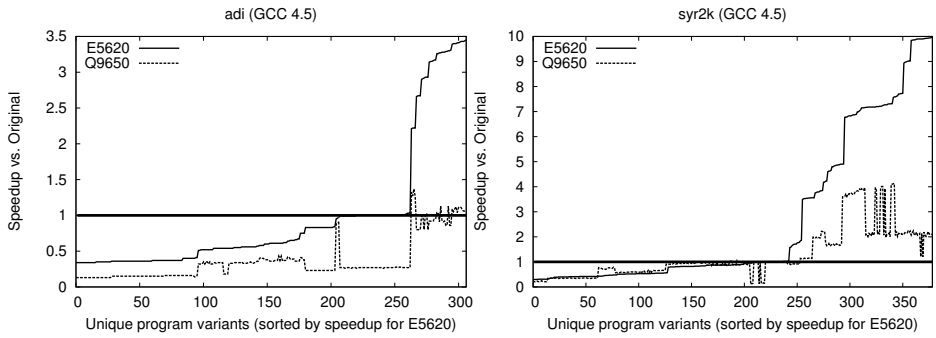

Fig. 5 Performance distribution for adi and syr2k using GCC 4.5

It is extremely interesting to observe that for both a stencil (adi) and a simple linear kernel (syr2k), the best performing variant for one processor has significantly lower performance on the other. We remark that despite being two Intel x86 64 bits processors, they significantly differ in design. The Q9650 is a Yorkfield dual-die quad-core design based on the Core 2 duo microarchitecture, with 12MB of L2 cache. The E5620 is a Westmere processor using the Nehalem microarchitecture, with hyper-threading and 12MB of L3 cache. This machine has two processors, and thus has a total of 16 H/W threads instead of 4 for the Q9650. And most notably,

Intel made significant changes on the E5620 adding a second-level branch predictor and translation lookaside buffer and using the Intel QuickPath Interconnect for data transfers in place of the slower Front Side Bus.

As a consequence, the relative impact of transformations such as vectorization, parallelism, or data locality is significantly changed. For instance, when there is a trade-off between data locality and parallelization, a better and faster data cache hierarchy in the Nehalem diminishes the need for our framework to focus on data locality improvement (e.g., tiling), but more available threads makes the need to perform effective parallelization for the Nehalem critical.

3.5 Sensitivity to Different Datasets

The best set of optimizations often depends on the program data, in particular when the input data can significantly change the instructions executed by the program. For instance, the sparsity of a matrix affects the number of floating point operations in a sparse-matrix computation. Previous work investigated the use of standardized datasets for arbitrary programs [14, 26], and their impact on compiler optimization.

The present work focuses exclusively on *static control programs*, where the control flow (and therefore the program instructions to be executed) can be fully determined at compile-time. That is, SCoPs are not sensitive to the dataset content: for a given input dataset size, the best optimization will be the same for any value of the elements of this dataset. This is a strong benefit of focusing exclusively on SCoPs, as we are not required to fine-tune the optimization sequence for different sets of inputs of identical size.

Dataset size

Despite a lack of sensitivity to the input data content, the total number of instructions the program will execute still heavily depends on the dataset *size*, that is the number of input data elements. We distinguish two features immediately depending on the dataset size: (1) the ability to keep (most of) the data in cache, for smaller datasets; and (2) the profitability of parallelization, when the cost of spawning threads is to be compared with the total number of instructions to be executed.

We illustrate the following with a matrix-multiply example. For the first case, we multiply matrices of size 8×8. The total dataset size for the three matrices (`double`) is 1.5kB, and the total number of floating point operations is 1024. The data fits fully in L1 data cache, and 512 SSE3 SIMD operations are required to execute the vector operations corresponding to the complete computation. In this case, program transformations such as tiling (for locality) and thread-level parallelization will clearly not improve the performance, because they address performance issues that are not seen for this program. Considering now the same program, but operating on matrices of size 1024×1024. The dataset size is now 25MB, and does not fit in the on-chip data cache. The overhead of spawning threads is negligible in comparison of the total number of operations to be executed. Therefore, optimizations such as tiling and thread-level parallelization become highly profitable.

The results presented in the current Section are limited to a single dataset size, the reference one for PolyBench 2.1 [30]. Depending on the benchmarks, the dataset size can be L2 resident, L3 resident, or most of the time larger than L3. That is, the benchmark set we have used spans most of the typical dataset size cases that can arise. The workload also differs vastly from benchmark to benchmark, as shown in Section 3.6 where for several benchmarks thread-level parallelization is not the optimal transformation.

We also remark that the machine learning techniques described in Section 4 use a hardware counters characterization of the input program to select the best transformation. Variation in the dataset size, and its implication in terms of data footprint and total workload to be executed are fully captured by hardware counters. Our approach to determine the best transformation is therefore robust to different dataset sizes, provided we use a training set that correctly spans the various key problem sizes associated with the profitability of each optimization. We remark that in the present work, we have limited our study to only one problem size per benchmark, focusing exclusively on out-of-L1 dataset sizes. These dataset sizes are representative of program which may benefit from agressive loop transformations, the target of the present article. Complementary techniques such as versioning can be used to find the optimal transformation for a few typical dataset sizes such as L1-resident, L2-resident, L3-resident and larger than L3. Such technique has been successfully used in ATLAS for instance [57] and is compatible with our approach.

3.6 Analysis of the Optimal Transformations

We conclude our analysis of the search space we consider by reporting, for both machines, the high-level optimizations that are used for the best performing optimized program variant obtained. Table 3 reports, for each benchmark and compiler, the optimizations turned on for the best performance on Nehalem. Table 4 shows the same data for the Q9650.

We observe interesting correlations between transformations. For instance, pre-transformations for vectorization is always beneficial (or at least, not detrimental) when using the SIMD-level parallelization pass. Wavefronting is useful for some stencils which are tiled, but not all. For instance, wavefronting is not beneficial on Nehalem for fdtd-apml. In contrast, wavefronting is always used in the best optimization for stencils on the Q9650 machine, together with tiling.

The benefit of rectangular tiling is demonstrated in numerous benchmarks, such as adi and dynprog for Q9650 or correlation and covariance for Nehalem. We observe that contrary to what is expected, tiling all loops which carry data reuse does not necessarily lead to the best performance. For instance, jacobi-1d-imper is not tiled, and for Nehalem, strip-mining the inner loop is the best choice for correlation and covariance. This result is counter-intuitive, and its observation is a contribution of our extensive analysis.

Building cost models for tiling is challenging, as illustrated by the difference in dimensions to be tiled between compilers and between architectures. On the other hand, as expected for Nehalem and for all but two benchmarks, thread-level paral-

	Fusion			Tiling			Parallelism		SIMD		Unroll-and-jam		
	M	S	N	T1	T2	T3	W	TLP	Pre	SLP	2	4	8
2mm	g			g	g	g		g/i	g	g			
3mm	g/i			g	g	g		g/i	g	g			
adi		g		g	g		g	g/i					
atax		g		g	g			g/i					g
bicg		g		g	g			g/i					g
cholesky	g	i						g/i					g
correlation	i	g				g		g/i	g/i	g/i			
covariance		g/i				g		g/i	g/i	g/i			
doitgen		g/i						g/i	g/i	g/i			
durbin	g/i							g/i					
dynprog													g/i
fdtd-2d								g/i					i
fdtd-apml		g/i			g			g/i				i	
gauss-filter								g/i					g/i
gemm		g/i		g	g	g		g/i	g	g			
gemver		g		g/i	g/i			g/i	g	g			g
gesummv		g/i		g/i	g			g/i					
gramschmidt	g/i			g/i				g/i					g
jacobi-1d-imper		g/i		g/i	g/i		g/i	g/i			g		
jacobi-2d-imper								g/i					
lu								g/i					
ludcmp	i	g						g/i					g
mvt	g			g	g			g/i					g
reg_detect		g						g/i	i	i		g	i
seidel		g/i			g	g	g	g/i	i	i	g		
symm	g	i		g				g/i			g/i		
syr2k		g/i						g/i					g
syrk		g						g/i			g		
trisolv		g/i		g/i	g/i			g/i	g/i	g/i	i		
trmm				i				i					g

Table 3 Summary of the high-level optimizations used to achieve the best performance on Intel E-5620, when using GCC 4.5 (g) or ICC 11.1 (i). For fusion, M corresponds to max-fuse, S to smart-fuse, and N to no-fuse. For tiling, T1 corresponds to tiling the first dimension by 32, T2 the second by 32, and T3 the third by 32. For parallelism, W corresponds to wavefronting and TLP to thread-level parallelization. For SIMD Pre corresponds to pre-vectorization and SLP to SIMD-level parallelization. For unroll-and-jam, 2, 4 and 8 correspond to the unrolling factors used.

lelization is part of the optimal transformation, and a naive heuristic could simply always apply this optimization. This was expected as the number of available threads on Nehalem is large (16 total), thus making coarse-grain parallelization a critical optimization for performance. Looking at Q9650 shows a very different trend, where for 7 out 30 benchmarks, thread-level parallelization is not optimal when using GCC. In addition, GCC 4.5 does not have automatic OpenMP parallelization features turned on in `-O3`, so it is clear that the compiler did not parallelize outer loops. This also confirms our hypothesis that on Q9650, data locality and instruction-level parallelism are more critical performance factors.

	Fusion			Tiling			Parallelism		SIMD		Unroll-and-jam		
	M	S	N	T1	T2	T3	W	TLP	Pre	SLP	2	4	8
2mm	g			g		g		g/i	g	g			
3mm	g/i							g/i	g	g			
adi		i		i	i		i	g/i					i
atax		i						i					g
bicg		i						i				g	
cholesky	g							g/i					g
correlation	i	g		i	i	i		g	g	g			
covariance		g		i	i	g/i		g	g	g			
doitgen		g/i		g	i	i		g/i	g/i	g/i			
durbin		i						g/i			i		
dynprog	i		g	i	i			i			i		
fdtd-2d		g		g/i	g/i		g/i	g/i			g		
fdtd-apml		g/i			g	i	g	g/i					
gauss-filter				i				g			i		
gemm		g/i		g	g/i	g		g/i	g	g			
gemver		g/i		g/i	g/i			g/i				g	
gesummv		i						g/i				g	
gramschmidt	i	g		g/i				g/i					
jacobi-1d-imper									i	i			g
jacobi-2d-imper		g/i			g	g	g/i	g/i			g/i		
lu		g		g	g		g	g/i	g	g			
ludcmp	g/i			g				g/i					g
mvt	g	i		g	g/i			g/i					g
reg_detect		i		g	g	g		g/i				g	i
seidel		g/i			g	g/i	g/i	g/i	i	i	g		
symm	g/i												
syr2k		g/i		g	g	g		g/i					
syrk		g/i		g	g	g		g/i					
trisolv								i	i	i			g/i
trmm				i				i					g

Table 4 Summary of the high-level optimizations used to achieve the best performance on Intel Q9650, when using GCC 4.5 (g) or ICC 10.1 (i). The descriptions of the column headings can be found in the caption of Table 3.

4 Selecting Effective Transformations

In this paper, we focus on the search of polyhedral optimizations with the highest impact as described in Section 2. When considering a space of semantics-preserving polyhedral optimizations, the optimization space can lead to a very large set of possible optimized versions of a program [44]. We achieved a tremendous reduction in the search space size using expert pruning when compared to these methods, but we still 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 optimizations to apply to a new program.

4.1 Characterization of Input Programs

In this work, we characterize the *dynamic* behavior of programs, by means of hardware performance counters. Using performance counters abstracts away the specifics of the machine, and overcomes the lack of precision and information of static characteristics. Also, models using performance counter characteristics of programs have been shown to out-perform models that use only static features of program [12].

A given program is represented by a feature vector of performance counters collected by running the program on the particular target machine. We use the PAPI library [38] to gather information about memory management, vectorization, and processor activity. In particular, for all cache 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 issued.

Category of PCs	List of PCs selected
Cache Line Access	CA-CLN, CA-ITV, CA-SHR
Level 1 Cache	L1-DCA, L1-DCH, L1-DCM, L1-ICA, L1-ICH, L1-ICM, L1-LDM, L1-STM, L1-TCA, L1-TCM
Level 2 & 3 Cache	L2-DCA, L2-DCM, L2-DCR, L2-DCW, L2-ICA, L2-ICH, L2-ICM, L2-LDM, L2-STM, L2-TCA, L2-TCH, L2-TCM, L2-TCR, L2-TCW, L3-TCA, L3-TCM
Branch Related	BR-CN, BR-INS, BR-MSP, BR-NTK, BR-PRC, BR-TKN, BR-UCN
Floating Point	DP-OPS, FDV-INS, FML-INS, FP-INS, FP-OPS, SP-OPS
Interrupt/Stall	HW-INT, RES-STL
TLB	TLB-DM, TLB-IM, TLB-SD, TLB-TL
Total Cycle or Instruction	TOT-CYC, TOT-IIS, TOT-INS
Load/Store Instruction	LD-INS, SR-INS
SIMD Instruction	VEC-DP, VEC-INS, VEC-SP

Table 5 Performance counters (PC): We collected 56 different performance counters available using PAPI library to characterize a program.

4.2 Speedup Prediction Model

A general formulation of the optimization problem is to construct a 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 [12, 21] 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.* [11, 20], but slightly adapted to support polyhedral optimizations 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 6 illustrates the speedup prediction model.

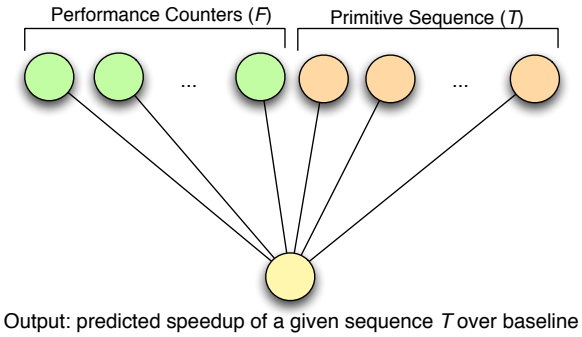

Fig. 6 Our speedup prediction model takes in two inputs. The first input is a characterization of the program consisting of a feature vector F of performance counters. The second input is a set of possible optimizations from the polyhedral optimization space. The target value for this predictor is the speedup of a specific optimization set T over baseline.

We implemented the speedup prediction model by using six different machine learning algorithms shown in Table 6 using Weka V3.6.2 [10]. For each machine learning algorithm, we used the default settings, except for support vector machines (SVM) and linear regression (LR). For the SVM and LR algorithms, we conducted experiments to tune parameters of those algorithms.

Name	Description
LR	Linear Regression
SVM	Support Vector Machine (Regression using Normalized Polynomial Kernel)
IBk	Instance-based Learning using K-Nearest Neighbor and Euclidean Distance
K*	Instance-based Learning using Entropic Distance
M5P	M5 Model Tree Based Learning
MLP	Multi-Layer Perceptron

Table 6 The six machine learning algorithms we evaluated in this paper.

The regression-based model demonstrates the relationship between dependent and independent variables, and we can use this model to observe the dependent variables according to the change of given independent variables. We used linear regression to fit a predictive model to a dependent variable which is speedup of programs, and independent variables which are performance counters and the polyhedral optimization sequence. Support vector machines (SVM) is a supervised machine learning algorithm, used for both classification and regression, and it can apply linear techniques to non-linear problems. First, an SVM algorithm transforms the training data into a linear space by using kernel functions, and then it uses a linear classifier to separate data with a hyperplane. SVMs not only finds a hyperplane to separate data, but it also finds the best hyperplane, i.e., the maximum margin hyperplane, that gives the largest separation of the data in the training examples from the set of all hyperplanes. IBk and K* are instance-based learning algorithms that predict the classification of

new instances based on instances already classified in the memory. These types of algorithms assume that similar instances belong to a similar class. IBk first selects the K-nearest neighbors of a new instance, then selects the class of the neighbor that is closest amongst them [2]. We used Euclidean distance function of all the features to find the nearest neighbors, but other distance functions can also be used, e.g., Manhattan distance. K* also selects from instances already classified. It then chooses the class of the predominant instance, but uses entropic distance. Entropic distance is defined as the complexity of transforming one instance into another one [15]. M5P generates M5 model trees, which look like conventional decision trees, but have linear regression functions at the leaf nodes. MLP (MultiLayer Perceptron) is neural network classifier, and we used back propagation to train the network.

4.3 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 machine learning algorithms used, we train one specific model for machine.

A model is trained as follows. For a given program P in the training set, (1) we compute its execution time E and collect its performance counters F; (2) for all possible sequences of polyhedral optimizations T_i, we 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) we 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 7.

For a new input program, we first collect a feature vector of performance counters from several runs of the program. Then, we use the model to predict the expected speedup of each set of optimizations T_i. By predicting the performance of each possible set, it is possible to rank them according to their expected speedup and select the set(s) with the greatest speedup. This is illustrated in Figure 8.

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 optimization set T_i in our polyhedral optimization space, given the characteristics of the unseen program. We order the predicted speedups to determine which optimization set is predicted best, and we apply the predicted best optimization set(s) to the unseen program.

Note in the experiments presented below, we use a *leave-one-benchmark-out cross-validation* procedure for evaluating our models. That is, the six models (LR, SVM, IBk, K*, M5P, or MLP) are trained on all program variants of each of the $N-1$ benchmarks and evaluated on the program variants of the benchmark that was left out. This procedure is repeated for each benchmark to be evaluated, that is, we construct a different model for each program in our training set.

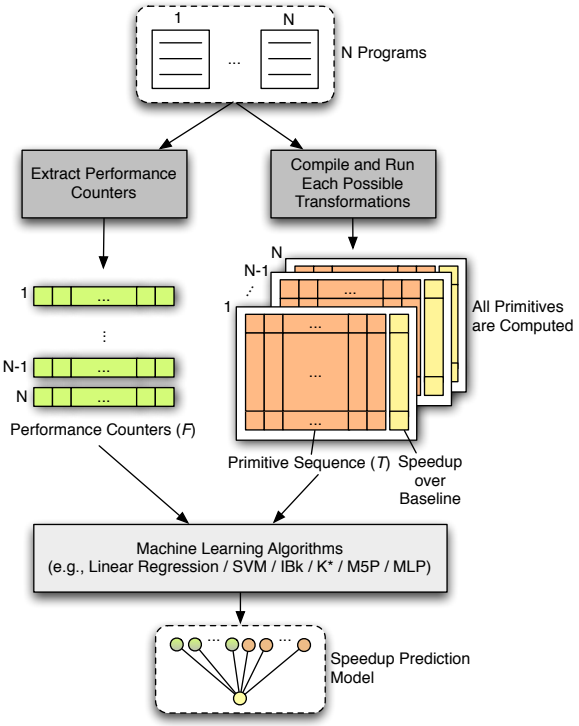

Fig. 7 To train the model, we collect performance counters F from each program in training set. We also collect the speedup for each optimization set T_i. We build the prediction model with F, T_i, and the associated speedup over baseline for T_i.

4.4 1-shot and Multi-model 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 a *1-shot model*.

It is worth considering an empirical evaluation of several candidate optimization sets, since a given model may not correctly predict the actual best optimization set for a program. A typical source of misprediction comes from the back-end compiler: depending on the input source code, the back-end compiler may perform different optimizations. For example, 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 11.1 is achieved when *no tiling* is applied by our polyhedral compiler. 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 compiled entirely differently by ICC, and as shown by our experiments even the same program is handled differently by ICC 11.1 and ICC 10.1 on two different machines.

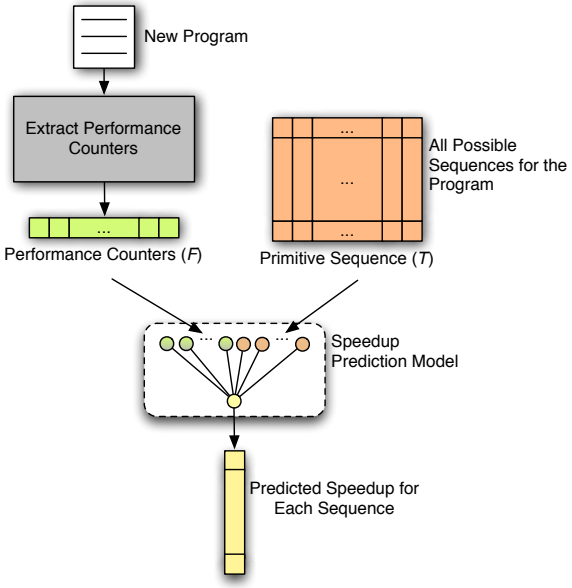

Fig. 8 To use the model, we collect performance counters F for a given new program. We give F and an optimization set T as input to model. The models produces a predicted speedup for each optimization set T_i. This prediction can be used in both an non-iterative and iterative scenario.

We propose to also evaluate an approach that combines the output of multiple individual models. In contrast to the 1-shot model, which does not require to run the transformed variant on the machine, we propose to resort to a small number of iterative compilation steps. We call this second scheme *multi-model*, since we use each optimization set that was predicted best by each individual 1-shot model. The optimization set which has the best observed performance is retained as the output of the compilation process.

5 Experimental Results

We now present extensive experimental results, using the platforms and benchmarks detailed in Section 3.

5.1 Training and Testing Time

We report in Table 7 the training and testing time for the 6 ML algorithms we have used. We recall that we have used Weka V3.6.2 [10], running on a Core2 Duo. The Training operates on all the program variants generated for a set of $N-1$ benchmarks, out of N benchmarks in our test suite. The Testing is done on all the variants of the benchmark which was left out during training. Because of different numbers of vari-

ants for each benchmark, we report the time range taken by the training and testing in Table 7.

	LR	SVM	IBk	K*	M5P	MLP
Training	3-10 mins	20-25 hours	10-20 mins	14-16 mins	13-15 mins	45-60 mins
Testing	1-4 secs	30-80 secs	5-10 secs	10-16 secs	1-12 secs	2-10 secs

Table 7 Weka V3.6.2 Training and Testing time, for the various ML algorithms we have used.

5.2 Evaluation of the Machine Learning Models on Nehalem

We show in Tables 8 and 9 the performance of the six different machine learning models we have evaluated using the 1-shot approach on the Nehalem machine. For each benchmark, we report the maximal performance improvement over the original code in the search space (Opt column). This improvement is relative to the performance obtained by running the native compiler on the original code. We also report the improvement obtained by a simple random method of generating an optimization set (averaging the performance of 100 distinct random draws). (Random column). Finally, we report the performance improvement achieved by the optimization set in our search space that is predicted best by each model (columns LR to MLP). The fraction (in percentage) of the optimal improvement is also reported.

Analysis

We observe that a very important performance improvement can be achieved with the polyhedral optimization sets we consider, on average a speedup of 9× for GCC and 5.6× for ICC, with peaks up to 28×. The range of the performance improvement is wide: with GCC (ICC) as the back-end compiler, 7 (12) benchmarks shows an improvement below 2×, while 10 (7) shows an improvement above 10×. We also observe that for all benchmarks, there exists at least a polyhedral optimization set that improves the performance.

Note that we have shown in Section 3 that for many benchmarks a vast majority of optimization sets can decrease performance. This is confirmed using Random where, choosing optimization sets randomly, we decrease performance compared to the original code for 11 benchmarks when using GCC (17 for ICC) as the backend compiler. Still, a simple random strategy increases performance on average by up to 2.5× using GCC. This is explained by the very large performance improvement that can be obtained for some benchmarks where the majority of transformed variants provide a solid improvement, such as 2mm or 3mm. Nevertheless, the fraction of possible improvement achieved by a simple strategy such as Random remains very low, i.e., around 25% for both compilers.

We observe that each of the six machine learning models we evaluated outperform Random, with K* reaching up to almost 7× performance improvement on average

Benchmark	Opt	Random	LR	SVM	IBk	K*	M5P	MLP
2mm	25.05×	5.84×(23%)	25.05×(100%)	25.05×(100%)	25.05×(100%)	25.05×(100%)	25.05×(100%)	25.05×(100%)
3mm	28.00×	5.25×(18%)	20.60×(73%)	28.00×(100%)	27.06×(96%)	28.00×(100%)	25.09×(89%)	19.40×(69%)
adi	3.45×	1.04×(30%)	2.94×(85%)	0.35×(10%)	2.22×(64%)	1.00×(29%)	2.22×(64%)	3.29×(95%)
atax	2.17×	0.41×(18%)	1.87×(86%)	0.12×(5%)	1.13×(51%)	0.18×(8%)	1.78×(81%)	0.91×(42%)
bicg	2.07×	0.48×(23%)	1.89×(91%)	0.95×(46%)	1.19×(57%)	1.17×(56%)	0.95×(45%)	0.95×(46%)
cholesky	1.14×	1.08×(94%)	1.14×(99%)	1.13×(99%)	1.14×(99%)	1.14×(99%)	1.14×(99%)	1.13×(99%)
correlation	19.25×	5.27×(27%)	10.03×(52%)	18.82×(97%)	18.96×(98%)	19.25×(100%)	18.11×(94%)	2.79×(14%)
covariance	20.98×	4.64×(22%)	10.22×(48%)	19.97×(95%)	20.01×(95%)	20.01×(95%)	15.15×(72%)	15.93×(75%)
doitgen	20.94×	5.06×(24%)	6.72×(32%)	18.23×(87%)	7.96×(38%)	5.85×(28%)	2.50×(11%)	7.96×(38%)
durbin	1.00×	0.99×(99%)	0.99×(99%)	0.98×(98%)	0.99×(99%)	0.99×(98%)	1.00×(99%)	0.99×(99%)
dynprog	0.94×	0.55×(58%)	0.84×(89%)	0.44×(47%)	0.58×(61%)	0.68×(72%)	0.69×(73%)	0.55×(58%)
fdtd-2d	5.17×	0.64×(12%)	0.80×(15%)	0.69×(13%)	5.10×(98%)	5.17×(100%)	5.10×(98%)	1.16×(22%)
fdtd-apml	8.21×	2.18×(26%)	3.48×(42%)	7.34×(89%)	6.74×(82%)	6.75×(82%)	7.36×(89%)	7.39×(90%)
gauss-filter	3.45×	1.04×(30%)	1.69×(49%)	2.15×(62%)	1.06×(30%)	1.05×(30%)	2.15×(62%)	2.14×(62%)
gemm	27.76×	10.21×(36%)	19.45×(70%)	17.95×(64%)	26.52×(95%)	26.52×(95%)	10.03×(36%)	26.52×(95%)
gemver	7.77×	1.92×(24%)	6.57×(84%)	6.57×(84%)	6.58×(84%)	6.58×(84%)	7.58×(97%)	7.77×(100%)
gesummv	2.31×	1.03×(44%)	1.59×(68%)	2.10×(90%)	2.16×(93%)	1.58×(68%)	2.18×(94%)	2.04×(88%)
gramschmidt	25.42×	9.74×(38%)	25.04×(98%)	6.83×(26%)	1.00×(3%)	6.80×(26%)	25.04×(98%)	21.47×(84%)
jacobi-1d	2.99×	0.53×(17%)	2.66×(88%)	1.00×(33%)	1.00×(33%)	0.46×(15%)	2.66×(88%)	0.07×(2%)
jacobi-2d	6.49×	1.06×(16%)	0.58×(8%)	1.91×(29%)	1.91×(29%)	6.49×(100%)	1.69×(26%)	2.78×(42%)
lu	6.20×	0.91×(14%)	0.92×(14%)	0.12×(1%)	6.20×(100%)	6.20×(100%)	3.81×(61%)	3.81×(61%)
ludcmp	1.10×	1.06×(96%)	1.04×(94%)	1.00×(91%)	1.10×(99%)	1.10×(99%)	1.10×(99%)	1.05×(96%)
mvt	13.10×	3.42×(26%)	11.38×(86%)	11.29×(86%)	11.33×(86%)	11.34×(86%)	6.66×(50%)	1.02×(7%)
reg-detect	1.91×	0.82×(42%)	0.59×(30%)	1.00×(52%)	0.24×(12%)	0.24×(12%)	0.58×(30%)	0.40×(21%)
seidel	7.10×	1.62×(22%)	0.82×(11%)	1.00×(14%)	1.00×(14%)	0.81×(11%)	0.92×(13%)	0.85×(12%)
symm	1.01×	1.00×(99%)	1.01×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)
syr2k	9.96×	1.99×(20%)	4.22×(42%)	1.76×(17%)	9.92×(99%)	9.94×(99%)	9.94×(99%)	1.00×(10%)
syrk	13.54×	2.83×(20%)	3.83×(28%)	0.76×(5%)	13.29×(98%)	13.42×(99%)	3.21×(23%)	1.29×(9%)
trisolv	2.01×	0.66×(32%)	1.89×(94%)	0.12×(5%)	1.20×(59%)	0.12×(6%)	1.20×(59%)	1.21×(60%)
trmm	1.35×	0.65×(48%)	0.78×(57%)	1.30×(96%)	1.35×(100%)	0.53×(39%)	1.35×(100%)	1.27×(94%)
Average	9.06×	2.46×(27%)	5.69×(62%)	6.00×(66%)	6.83×(75%)	6.98×(77%)	6.24×(68%)	5.44×(60%)

Table 8 This table shows performance improvement on the Intel Xeon E5620 (baseline: GCC 4.5 -O3) for the 1-shot model. Each of six machine learning models outperforms Random for each of the benchmarks. We achieved up to 77% of Opt on average by using the K* model. We also observe that there is no one machine learning algorithm that is the best for all benchmarks. For example, although K* gives the best performance improvements for most benchmarks, MLP produces the best model for adi and SVM produces the best model for doitgen.

with GCC and with IBk reaching to 3.73× with ICC. On average, both K* and IBk are top two best models for both compilers. A very interesting observation is that none of the models, including K*, performs consistently best for all benchmarks. Table 9 shows the ability of each model to successfully predict the best optimization set for at least some benchmarks, while other models fail. That is, each model produces the best improvement on at least one benchmark. As an illustration, K* using GCC on gramschmidth does not provide an improvement, while LR produces an almost optimal variant reaching 25.04× improvement (98% of the maximal improvement possible for this benchmark). Even MLP, the worst-performing model on average for GCC, succeeds in finding the optimal variant for gemver, while all other models fail at doing so. The effectiveness of the models on average differs depending on the backend compiler used. For example, MLP using ICC performs well (it is the third best model on average), while it is the worst model for GCC. Nevertheless LR and SVM perform on average consistently worse than IBk and M5P, and K* performs consistently best. We conclude this analysis by observing that for both compilers, no model is able to guarantee, in one shot, to not decrease the performance of the original code. K* significantly decreases the performance for 5 benchmarks for GCC and 6 for ICC, while SVM decreases the performance for 11 benchmarks for both compilers.

Benchmark	Opt	Random	LR	SVM	IBk	K*	M5P	MLP
2mm	13.37×	1.42×(10%)	6.43×(48%)	4.44×(33%)	12.84×(96%)	13.37×(100%)	7.80×(58%)	12.84×(96%)
3mm	13.38×	1.07×(8%)	3.98×(29%)	4.33×(32%)	13.04×(97%)	13.35×(99%)	7.63×(57%)	13.38×(100%)
adi	3.80×	0.69×(18%)	2.56×(67%)	2.57×(67%)	1.09×(28%)	2.60×(68%)	2.60×(68%)	2.60×(68%)
atax	2.57×	0.63×(24%)	1.95×(76%)	2.56×(99%)	2.18×(84%)	2.56×(99%)	1.17×(45%)	1.17×(45%)
bicg	1.71×	0.50×(29%)	1.30×(76%)	1.45×(85%)	1.45×(85%)	1.45×(85%)	0.51×(29%)	1.30×(76%)
cholesky	1.04×	0.67×(64%)	1.04×(99%)	0.59×(57%)	1.01×(97%)	1.02×(97%)	0.58×(55%)	1.01×(97%)
correlation	8.47×	1.99×(23%)	4.88×(57%)	7.91×(93%)	8.47×(100%)	8.29×(97%)	1.99×(23%)	8.29×(97%)
covariance	9.03×	1.96×(21%)	4.99×(55%)	9.03×(100%)	8.87×(98%)	9.03×(100%)	8.87×(98%)	8.78×(97%)
doitgen	12.89×	2.99×(23%)	3.32×(25%)	0.91×(7%)	12.58×(97%)	12.58×(97%)	10.27×(79%)	3.26×(25%)
durbin	1.00×	0.99×(98%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	0.99×(98%)
dynprog	1.14×	0.70×(61%)	0.98×(86%)	0.70×(61%)	0.45×(39%)	0.54×(47%)	0.62×(54%)	0.54×(47%)
fdtd-2d	1.03×	0.09×(9%)	0.18×(17%)	0.02×(2%)	1.01×(98%)	0.13×(12%)	0.21×(20%)	0.13×(12%)
fdtd-apml	5.96×	1.62×(27%)	1.58×(26%)	0.90×(15%)	5.83×(97%)	1.58×(26%)	1.57×(26%)	1.76×(29%)
gauss-filter	8.48×	2.63×(31%)	3.69×(43%)	4.41×(52%)	3.40×(40%)	3.40×(40%)	4.41×(52%)	4.18×(49%)
gemm	13.14×	2.42×(18%)	4.28×(32%)	2.31×(17%)	7.49×(57%)	11.31×(86%)	4.35×(33%)	0.88×(6%)
gemver	1.74×	0.68×(39%)	1.47×(84%)	0.80×(45%)	1.74×(100%)	0.71×(41%)	0.85×(48%)	0.79×(45%)
gesummv	1.05×	0.52×(49%)	0.85×(81%)	0.99×(94%)	0.99×(94%)	0.99×(94%)	0.83×(79%)	0.99×(94%)
gramschmidt	22.27×	7.77×(34%)	1.76×(7%)	7.18×(32%)	17.77×(79%)	7.05×(31%)	22.27×(100%)	21.59×(96%)
jacobi-1d	9.07×	1.13×(12%)	9.07×(100%)	0.98×(10%)	0.98×(10%)	0.98×(10%)	0.98×(10%)	0.98×(10%)
jacobi-2d	4.18×	0.62×(14%)	0.42×(10%)	0.60×(14%)	0.42×(10%)	0.52×(12%)	0.52×(12%)	0.40×(9%)
lu	4.72×	0.94×(20%)	0.85×(18%)	0.37×(7%)	0.13×(2%)	0.23×(4%)	0.13×(2%)	4.00×(84%)
ludcmp	1.03×	1.00×(96%)	1.03×(99%)	1.03×(100%)	1.01×(98%)	1.03×(99%)	1.01×(98%)	1.01×(98%)
mvt	2.06×	0.72×(35%)	1.54×(74%)	0.76×(36%)	2.06×(100%)	2.06×(100%)	1.54×(74%)	0.69×(33%)
reg-detect	1.42×	0.66×(46%)	0.49×(34%)	0.35×(24%)	1.03×(72%)	0.98×(69%)	0.91×(64%)	0.38×(26%)
seidel	12.34×	2.45×(19%)	1.05×(8%)	8.48×(68%)	1.00×(8%)	1.00×(8%)	1.21×(9%)	1.01×(8%)
symm	1.01×	1.00×(99%)	1.00×(98%)	1.01×(100%)	1.01×(100%)	1.01×(100%)	1.01×(100%)	1.00×(98%)
syr2k	1.00×	0.20×(20%)	0.42×(42%)	1.00×(99%)	1.00×(100%)	1.00×(100%)	1.00×(99%)	0.75×(74%)
syrk	1.05×	0.19×(17%)	0.19×(18%)	1.05×(100%)	1.05×(99%)	1.05×(100%)	0.16×(15%)	1.05×(99%)
trisolv	2.74×	0.67×(24%)	0.67×(24%)	0.67×(24%)	0.16×(5%)	0.67×(24%)	2.72×(99%)	0.59×(21%)
trmm	5.38×	0.94×(17%)	0.81×(15%)	1.01×(18%)	0.71×(13%)	1.01×(18%)	1.01×(18%)	0.81×(15%)
Average	5.60×	1.33×(23%)	2.13×(38%)	2.31×(41%)	3.73×(66%)	3.42×(61%)	2.99×(53%)	3.24×(57%)

Table 9 This table shows performance improvement on the Intel Xeon E5620 (baseline: ICC 11.1 -fast) for the 1-shot model. This table shows similar results to Table 8. Each of six machine learning algorithms outperforms Random, reaching up to 3.7× speedup with IBk model. Interestingly, we notice that the MLP model performs well for ICC while it was the worst model for GCC.

This motivates an approach we call the multi-model approach evaluated in Section 5.4, which combines the predictions of multiple models to select a transformation. We show in Section 5.4 that our multi-model approach decreases the performance for only one benchmark on Nehalem, for both compiler, while providing substantial performance improvements for the other benchmarks.

5.3 Evaluation of the Machine Learning Models on Core 2 Quad

Tables 10 and 11 report, for all benchmarks, the performance of the models we have evaluated in a similar fashion as in the previous section, but for the Q9650 machine.

Analysis

Similarly to the Nehalem machine, we observe that for most benchmarks and compilers there exists at least one polyhedral transformation that achieves a performance significantly better to the original code. Nevertheless the average maximal performance improvement in our search space is lower with the Q9650 machine than on the Nehalem. The Q9650 has a lower number of computing units available (less cores), and thus the maximal improvement obtained by exploiting parallelism is reduced. Still, a significant improvement is observed, from 3.7× on average for GCC to 4.6×

Benchmark	Opt	Random	LR	SVM	IBk	K*	M5P	MLP
2mm	5.82×	2.14×(36%)	5.24×(90%)	0.50×(8%)	1.97×(33%)	1.98×(34%)	1.98×(34%)	1.97×(33%)
3mm	10.55×	1.93×(18%)	8.06×(76%)	8.02×(76%)	9.36×(88%)	7.60×(72%)	1.94×(18%)	4.01×(38%)
adi	1.37×	0.40×(29%)	0.80×(58%)	1.00×(72%)	1.05×(76%)	1.27×(92%)	0.80×(58%)	1.05×(76%)
atax	1.02×	0.33×(32%)	0.48×(47%)	0.25×(24%)	1.00×(97%)	1.00×(97%)	0.17×(17%)	0.58×(57%)
bicg	1.04×	0.24×(23%)	0.39×(37%)	1.00×(96%)	0.39×(37%)	1.00×(96%)	0.55×(52%)	0.35×(33%)
cholesky	1.05×	1.03×(97%)	1.02×(97%)	1.01×(96%)	1.02×(97%)	1.02×(97%)	1.02×(97%)	1.02×(97%)
correlation	14.37×	4.03×(28%)	7.37×(51%)	0.88×(6%)	13.91×(96%)	13.10×(91%)	13.91×(96%)	7.17×(49%)
covariance	15.43×	6.36×(41%)	7.76×(50%)	14.44×(93%)	8.03×(52%)	7.38×(47%)	14.44×(93%)	14.44×(93%)
doitgen	5.75×	2.27×(39%)	4.58×(79%)	4.50×(78%)	4.50×(78%)	0.66×(11%)	2.05×(35%)	4.16×(72%)
durbin	1.00×	0.97×(97%)	0.96×(95%)	0.95×(95%)	0.99×(98%)	0.99×(98%)	0.98×(98%)	0.97×(97%)
dynprog	0.99×	0.55×(55%)	0.28×(28%)	0.36×(36%)	0.80×(80%)	0.22×(21%)	0.46×(46%)	0.19×(19%)
fdtd-2d	3.78×	1.62×(42%)	1.86×(49%)	2.98×(78%)	2.98×(78%)	1.07×(28%)	1.51×(39%)	3.78×(100%)
fdtd-apml	3.53×	1.66×(46%)	2.05×(58%)	2.97×(84%)	0.76×(21%)	2.71×(76%)	2.54×(72%)	1.11×(31%)
gauss-filter	1.44×	0.58×(40%)	0.84×(58%)	0.46×(32%)	1.41×(98%)	0.46×(31%)	0.83×(57%)	0.58×(40%)
gemm	4.21×	1.96×(46%)	3.94×(93%)	3.94×(93%)	3.89×(92%)	3.84×(91%)	3.21×(76%)	3.87×(92%)
gemver	2.21×	1.01×(45%)	1.66×(75%)	0.20×(9%)	1.74×(78%)	1.74×(78%)	1.36×(61%)	1.60×(72%)
gesummv	1.88×	0.78×(41%)	1.02×(54%)	1.08×(57%)	1.08×(57%)	1.39×(74%)	1.42×(75%)	1.41×(75%)
gramschmidt	7.95×	2.39×(30%)	5.24×(66%)	2.89×(36%)	1.00×(12%)	1.19×(14%)	5.79×(72%)	5.85×(73%)
jacobi-1d	1.02×	0.19×(18%)	0.17×(16%)	0.07×(7%)	0.32×(31%)	0.19×(18%)	0.19×(18%)	0.17×(17%)
jacobi-2d	2.89×	1.21×(41%)	1.46×(50%)	2.28×(78%)	1.35×(46%)	1.53×(52%)	1.32×(45%)	1.29×(44%)
lu	3.37×	0.30×(9%)	1.12×(33%)	0.17×(5%)	1.16×(34%)	1.16×(34%)	3.36×(99%)	0.16×(4%)
ludcmp	1.01×	1.00×(99%)	1.00×(99%)	0.99×(97%)	1.00×(98%)	1.01×(99%)	1.00×(98%)	1.00×(98%)
mvt	2.34×	1.32×(56%)	2.01×(86%)	0.62×(26%)	2.19×(93%)	1.77×(76%)	2.01×(86%)	2.01×(86%)
reg-detect	2.50×	0.98×(39%)	1.10×(44%)	0.22×(8%)	0.22×(8%)	1.12×(44%)	1.12×(44%)	0.34×(13%)
seidel	2.83×	1.11×(39%)	0.85×(30%)	0.84×(29%)	1.00×(35%)	2.38×(83%)	0.69×(24%)	0.85×(30%)
symm	1.01×	0.99×(98%)	1.00×(99%)	0.99×(98%)	1.00×(99%)	1.00×(99%)	1.00×(99%)	1.00×(99%)
syr2k	4.14×	1.29×(31%)	2.00×(48%)	1.66×(40%)	2.22×(53%)	2.08×(50%)	1.01×(24%)	2.13×(51%)
syrk	3.58×	1.35×(37%)	1.97×(55%)	2.71×(75%)	1.24×(34%)	2.86×(79%)	1.22×(34%)	1.24×(34%)
trisolv	1.00×	0.57×(57%)	0.45×(45%)	1.00×(99%)	0.46×(46%)	0.86×(85%)	0.99×(99%)	0.64×(63%)
trmm	1.02×	0.59×(57%)	0.60×(59%)	0.63×(61%)	0.62×(60%)	0.65×(63%)	0.65×(63%)	0.59×(57%)
Average	3.67×	1.37×(37%)	2.24×(61%)	1.99×(54%)	2.29×(62%)	2.17×(59%)	2.32×(63%)	2.18×(59%)

Table 10 The table describes the performance improvement for the Intel Quad Q9650 (baseline: GCC 4.4 -O3) for our 1-Shot model. We observe that each of machine learning models outperforms Random. Unlike the Nehalem results, we observe that the K* model is not the best model. Although the K* model still gives good performance overall, IBk and M5P outperforms K* on average while reaching a speedup of up to 2.3×.

for ICC, with peaks up to 23×. The number of benchmarks for which there is little to no improvement increased from Nehalem with up to 13 benchmarks having a speedup lower than 2× for GCC.

Interestingly, comparing to the Nehalem, K* is not the best performing model on average for the Q9650. M5P and IBk perform consistently better on average, and they also reduce the number of benchmarks for which the performance is decreased by the selected transformation. We also notice for the Q9650 that each model performs well on a specific benchmark while the other models fail for that benchmark. For example, LR chooses an optimization set that reaches 90% of the maximal improvement for 2mm, while all other models fail to discover more than 34% of the maximal improvement for GCC. Similarly, MLP is the only model that finds the optimal optimization set for fdtd-2d. This pattern is observed for several benchmarks.

We conclude this analysis by observing that similarly to the Nehalem results, the models can decrease the performance of several benchmarks relative to not optimizing the benchmarks with PoCC. For example, M5P decreases the performance for seven benchmarks for both compilers. In contrast, the multi-model approach (discussed next) significantly reduces the number of benchmarks it obtains worse performance on than not using PoCC.

Benchmark	Opt	Random	LR	SVM	IBk	K*	M5P	MLP
2mm	19.51×	2.14×(11%)	8.65×(44%)	8.69×(44%)	19.51×(100%)	8.84×(45%)	7.73×(39%)	8.56×(43%)
3mm	24.75×	1.93×(7%)	11.31×(45%)	8.85×(35%)	24.68×(99%)	24.68×(99%)	24.75×(100%)	24.75×(100%)
adi	1.89×	0.40×(21%)	1.32×(69%)	0.31×(16%)	0.94×(49%)	0.23×(12%)	0.60×(32%)	0.27×(14%)
atax	2.55×	0.33×(13%)	1.32×(51%)	1.28×(50%)	2.10×(82%)	1.79×(70%)	1.00×(39%)	1.83×(71%)
bicg	1.57×	0.24×(15%)	0.79×(50%)	0.79×(50%)	1.02×(64%)	1.57×(100%)	1.12×(70%)	1.52×(96%)
cholesky	1.10×	1.03×(93%)	1.09×(99%)	1.10×(99%)	1.10×(100%)	0.61×(56%)	1.09×(99%)	1.02×(92%)
correlation	6.63×	4.03×(60%)	4.10×(61%)	5.86×(88%)	6.63×(100%)	2.67×(40%)	6.63×(100%)	4.92×(74%)
covariance	5.24×	6.36×(121%)	3.28×(62%)	3.69×(70%)	4.22×(80%)	5.24×(100%)	3.58×(68%)	2.92×(55%)
doitgen	2.86×	2.27×(79%)	0.80×(27%)	2.05×(71%)	2.05×(71%)	0.80×(28%)	2.09×(73%)	1.19×(41%)
durbin	1.06×	0.97×(91%)	0.77×(72%)	1.01×(95%)	1.00×(94%)	1.00×(94%)	1.04×(98%)	0.77×(72%)
dynprog	1.76×	0.55×(31%)	0.15×(8%)	0.21×(12%)	0.18×(10%)	0.20×(11%)	0.23×(13%)	0.18×(10%)
fdtd-2d	3.17×	1.62×(51%)	2.56×(81%)	0.80×(25%)	1.03×(32%)	0.81×(25%)	1.08×(34%)	0.81×(25%)
fdtd-apml	2.69×	1.66×(61%)	0.53×(19%)	2.63×(98%)	2.63×(98%)	1.36×(50%)	0.35×(13%)	0.59×(21%)
gauss-filter	4.55×	0.58×(12%)	2.01×(44%)	1.30×(28%)	2.20×(48%)	1.22×(26%)	2.85×(62%)	2.05×(45%)
gemm	2.82×	1.96×(69%)	1.40×(49%)	1.11×(39%)	2.50×(88%)	2.53×(89%)	2.53×(89%)	1.62×(57%)
gemver	1.47×	1.01×(68%)	1.22×(83%)	1.22×(83%)	0.79×(53%)	0.79×(53%)	0.91×(61%)	0.94×(63%)
gesummv	1.75×	0.78×(44%)	0.66×(38%)	0.80×(45%)	1.37×(78%)	1.43×(82%)	0.66×(38%)	0.80×(45%)
gramschmidt	22.94×	2.39×(10%)	12.09×(52%)	4.89×(21%)	1.01×(4%)	1.01×(4%)	7.35×(32%)	4.79×(20%)
jacobi-1d	4.71×	0.19×(4%)	2.87×(60%)	3.18×(67%)	0.87×(18%)	2.81×(59%)	3.18×(67%)	2.81×(59%)
jacobi-2d	6.23×	1.21×(19%)	5.44×(87%)	4.02×(64%)	1.25×(20%)	3.02×(48%)	2.72×(43%)	2.72×(43%)
lu	3.50×	0.30×(8%)	0.23×(6%)	3.50×(100%)	0.84×(24%)	0.72×(20%)	0.84×(24%)	0.84×(24%)
ludcmp	0.98×	1.00×(102%)	0.98×(99%)	0.96×(97%)	0.98×(99%)	0.98×(99%)	0.98×(99%)	0.96×(97%)
mvt	1.92×	1.32×(68%)	1.31×(68%)	1.02×(52%)	1.56×(81%)	1.56×(81%)	1.18×(61%)	0.88×(45%)
reg-detect	1.12×	0.98×(87%)	1.03×(91%)	0.92×(82%)	0.61×(54%)	0.82×(73%)	0.62×(55%)	0.98×(87%)
seidel	2.37×	1.11×(47%)	0.95×(39%)	1.63×(68%)	1.00×(42%)	1.18×(49%)	1.05×(44%)	1.00×(42%)
symm	1.02×	0.99×(96%)	1.02×(99%)	1.02×(99%)	1.01×(99%)	1.00×(98%)	1.00×(98%)	1.02×(99%)
syr2k	1.03×	1.29×(125%)	0.03×(2%)	1.01×(98%)	1.01×(98%)	0.19×(18%)	0.11×(11%)	0.28×(27%)
syrk	1.08×	1.35×(124%)	0.18×(16%)	0.86×(79%)	0.86×(79%)	0.97×(89%)	0.97×(89%)	0.84×(77%)
trisolv	3.15×	0.57×(18%)	0.52×(16%)	1.30×(41%)	1.00×(31%)	0.51×(16%)	0.49×(15%)	1.00×(31%)
trmm	1.56×	0.59×(37%)	1.56×(100%)	0.99×(63%)	0.99×(63%)	0.67×(42%)	0.78×(49%)	0.99×(63%)
Average	4.57×	1.37×(30%)	2.34×(51%)	2.23×(48%)	2.90×(63%)	2.37×(52%)	2.65×(58%)	2.46×(53%)

Table 11 The table describes the performance improvement for the Intel Quad Q9650 (baseline: ICC 10.1 -fast) for our 1-Shot model. We see similar results as in Table 10. Random is not able to outperform any of the six machine learning models. The M5P, IBk, and MLP models give performance improvements for most programs reaching up to 2.9×. Also, contrary to the results on the Nehalem architecture, the M5P or IBk model is the best model on the Q9650 architecture.

5.4 Multi-model Evaluation

Table 12 reports the performance achieved by the multi-model approach for each benchmark on each architecture/compiler pair. In contrast to the 1-shot approach, which is a compile-time only approach, our multi-model approach requires six optimized variants of the code to be executed on the target machine, corresponding to the predicted optimization configurations from each of the six individual models. That is, we optimize a benchmark with the optimization configuration predicted to give the best speedup from each machine learning model. Then, we evaluate each of these optimized variants of the benchmark and return the variant that gives the best speedup.

Analysis

We observe in Table 12 that our multi-model approach significantly outperforms the 1-shot approach, for all architecture/ compiler pairs we experimented with. In terms of average performance improvement, the multi-model consistently discovers more than 70% of the optimal improvement, a jump from 58% for ICC on Q9650. More importantly, for the majority of the benchmarks for Nehalem, the optimal (or close to optimal) variant is discovered by the multi-model. For the Q9650, the optimal variant is discovered by the multi-model approach for more than a third of the benchmarks.

Benchmark	Nehalem-gcc	Nehalem-icc	Q9650-gcc	Q9650-icc
2mm	25.05× (100.00%)	13.37× (100.00%)	5.24× (90.00%)	14.36× (73.00%)
3mm	28.00× (100.00%)	13.38× (100.00%)	9.32× (88.00%)	24.75× (100.00%)
adi	3.45× (100.00%)	2.60× (68.00%)	1.27× (92.00%)	1.32× (69.00%)
atax	1.87× (86.00%)	2.57× (100.00%)	1.00× (98.00%)	2.55× (100.00%)
bicg	1.89× (91.00%)	1.45× (84.00%)	1.04× (100.00%)	1.57× (100.00%)
cholesky	1.14× (100.00%)	1.04× (100.00%)	1.03× (98.00%)	1.10× (100.00%)
correlation	19.25× (100.00%)	8.29× (97.00%)	13.91× (96.00%)	6.63× (100.00%)
covariance	20.98× (100.00%)	9.03× (100.00%)	14.44× (93.00%)	5.24× (100.00%)
doitgen	18.23× (87.00%)	12.58× (97.00%)	4.69× (81.00%)	2.09× (73.00%)
durbin	1.00× (100.00%)	1.00× (100.00%)	0.99× (99.00%)	1.04× (98.00%)
dynprog	0.84× (89.00%)	0.98× (85.00%)	0.86× (86.00%)	0.23× (13.00%)
fdtd-2d	5.17× (100.00%)	0.21× (20.00%)	3.15× (83.00%)	2.56× (80.00%)
fdtd-apml	7.36× (89.00%)	1.76× (29.00%)	3.32× (94.00%)	2.63× (97.00%)
gauss-filter	2.15× (62.00%)	4.41× (52.00%)	1.43× (99.00%)	2.85× (62.00%)
gemm	27.62× (99.00%)	11.31× (86.00%)	3.94× (93.00%)	2.53× (89.00%)
gemver	7.77× (100.00%)	1.47× (84.00%)	1.74× (78.00%)	1.25× (85.00%)
gesummv	2.18× (94.00%)	0.99× (94.00%)	1.42× (75.00%)	1.43× (81.00%)
gramschmidt	25.04× (98.00%)	22.27× (100.00%)	5.80× (72.00%)	12.09× (52.00%)
jacobi-1d	2.66× (88.00%)	9.07× (100.00%)	1.02× (100.00%)	3.68× (78.00%)
jacobi-2d	6.49× (100.00%)	1.58× (37.00%)	2.28× (78.00%)	5.44× (87.00%)
lu	6.20× (100.00%)	4.72× (100.00%)	3.36× (99.00%)	3.50× (100.00%)
ludcmp	1.10× (100.00%)	1.03× (100.00%)	1.01× (100.00%)	0.98× (100.00%)
mvt	11.38× (86.00%)	2.06× (100.00%)	2.19× (93.00%)	1.56× (81.00%)
reg-detect	1.33× (69.00%)	0.98× (69.00%)	1.59× (63.00%)	1.03× (91.00%)
seidel	5.16× (72.00%)	10.14× (82.00%)	2.38× (84.00%)	1.63× (68.00%)
symm	1.01× (100.00%)	1.01× (100.00%)	1.00× (99.00%)	1.02× (100.00%)
syr2k	9.94× (99.00%)	1.00× (100.00%)	2.15× (51.00%)	1.01× (98.00%)
syrk	13.44× (99.00%)	1.05× (100.00%)	2.93× (81.00%)	0.97× (89.00%)
trisolv	1.89× (94.00%)	2.72× (99.00%)	1.00× (100.00%)	1.86× (59.00%)
trmm	1.35× (100.00%)	1.01× (18.00%)	0.65× (63.00%)	1.56× (100.00%)
Average	8.70× (93.61%)	4.84× (83.64%)	3.20× (87.94%)	3.68× (84.42%)

Table 12 This table shows the performance improvement of using the multi-model approach for all configurations. The multi-model compiles the program using the predicted best optimization configuration from each individual model and keeps the best performing one. Thus, six optimized program variants on the machine are evaluated in total.

The multi-model approach also achieves very significant improvements in terms performance guarantees: for all architecture/compiler pairs, the multi-model approach selects a variant that does not degrade performance for all but two benchmarks at most. However, there are still some benchmarks that see a degradation in performance with the multi-model approach. For example, dynprog sees its performance consistently degraded for all architecture/compiler pairs. This is a benchmark for which there is very little performance improvement to be discovered in the search space. We also see a degradation for trmm on the Q9650 using GCC.

In addition, we observe that our multi-model approach on Nehalem using ICC fails to discover good performance for fdtd-2d, but also to a lesser extent with jacobi-2d. This is consistent with the 1-shot results shown before. We suspect this is because for these two stencil codes, in contrast to other benchmarks, tiling is not part of the optimal transformation (shown in Table 3). So it is likely that the models output a

tiled variant, which for this very specific architecture/compiler configuration does not lead to the best performance.

Algorithms	Nehalem-gcc	Nehalem-icc	Q9650-gcc	Q9650-icc
LR (6-shot)	6.00× (68.5%)	2.48× (54.7%)	2.38× (66.0%)	2.61× (62.1%)
SVM (6-shot)	6.49× (64.4%)	3.48× (65.8%)	2.26× (64.0%)	2.97× (68.4%)
IBk (6-shot)	7.08× (76.3%)	4.08× (75.7%)	2.51× (70.0%)	3.03× (72.1%)
K* (6-shot)	7.56× (83.4%)	4.05× (70.4%)	2.68× (77.0%)	2.64× (63.4%)
M5P (6-shot)	6.45× (75.2%)	3.06× (57.6%)	2.58× (72.1%)	3.06× (70.9%)
MLP (6-shot)	6.35× (68.1%)	3.74× (69.1%)	2.59× (67.2%)	2.79× (69.1%)
Multi-model	8.70× (93.61%)	4.84× (83.64%)	3.20× (87.94%)	3.68× (84.42%)

Table 13 This table shows the average performance improvement of the multi-model approach and the 6-shot models for all configurations. We observe that multi-model approach outperforms any of 6-shot machine learning models and achieves up to 94% of the maximum available speedup in our search space.

Table 13 shows that average performance improvements with our multi-model approach can outperform all models using six evaluations, which we term the 6-shot approach. That is, using our multi-model approach (which uses six predicted optimization configuration, one from each model), we can achieve better performance improvements than by using the top six predicted optimization configurations from any one single model.

5.5 Evaluation of Feature Selection

In this section, we evaluate our models using a subset of performance counters deemed to be the most predictive. The main benefit of using a subset of counters is that we can reduce the number of application runs it takes to characterize that application. Using the full set of performance counters available on an architecture can take a large number of program executions to collect, especially if multiplexing [2] is not used. In addition, training time can be reduced if the number of performance counters collected is reduced. To find an effective subset of performance counters to use, we analyzed the output of LR models trained using greedy attribution selection mode. The output of the model lists the performance counters that were most informative in building the model. For each architecture/compiler configuration, we used the subset of performance counters that were used to build the LR model. As a result, we used 2 to 5 performance counters depending on the architecture/compiler configuration as shown in Table 14. Each configuration had a unique subset of important performance counters, including counters for different cache levels, TLB statistics, and instruction types.

Using the six different machine learning algorithms we evaluated for this research, we retrained our models with the subset of performance counters deemed important by the LR model, and we compared these models to using the full set of performance counters available on each architecture. Table 15 shows results from

[2] Note that multiplexing reduces the accuracy of the performance counter information collected.

Machine-Compiler	List of performance counters used for all models
Nehalem GCC4.5	BR-MSP, L1-ICA
Nehalem ICC11.1	L2-ICM, L3-TCM, TLB-IM, TLB-SD
Q9650 GCC4.4	L2-STM, L2-ICA
Q9650 ICC 10.1	BR-CN, BR-NTK, TOT-IIS, SR-INS, VEC-DP

Table 14 This table shows the 2 to 5 performance counters that were picked by the LR model for each architecture/compiler configuration. Note that the full set of performance counters is 38 for Nehalem and 49 for Q9650.

evaluating each of the models in a 1-shot scenario, and Table 16 shows results for our multi-model scenario using the subset of performance counters.

Number of PCs	LR	SVM	IBk	K*	M5P	MLP	OPT
Intel Xeon E5620 (Baseline: GCC 4.5 -O3)							
2	5.69×(62.8%)	4.84×(53.4%)	6.13×(67.7%)	7.35×(81.1%)	6.47×(71.4%)	5.11×(56.4%)	9.06×
38	5.69×(62.8%)	6.00×(66.2%)	6.22×(68.7%)	6.98×(77.0%)	6.24×(68.9%)	5.26×(58.1%)	9.06×
Intel Xeon E5620 (Baseline: ICC 11.1 -fast)							
4	2.13×(38.0%)	2.36×(42.1%)	3.40×(60.7%)	3.44×(61.4%)	3.02×(53.9%)	2.83×(50.5%)	5.60×
38	2.13×(38.0%)	2.31×(41.2%)	3.01×(53.8%)	3.42×(61.1%)	2.99×(53.4%)	3.19×(57.0%)	5.60×
Intel Quad Q9650 (Baseline: GCC 4.4 -O3)							
2	2.22×(60.5%)	1.74×(47.4%)	1.70×(46.3%)	2.68×(73.0%)	2.70×(73.6%)	1.94×(52.9%)	3.67×
49	2.24×(61.0%)	1.99×(54.2%)	2.33×(63.5%)	2.17×(59.1%)	2.32×(63.2%)	2.09×(56.9%)	3.67×
Intel Quad Q9650 (Baseline: ICC 10.1 -fast)							
5	2.34×(51.2%)	2.57×(56.2%)	2.16×(47.3%)	2.27×(49.7%)	1.78×(38.9%)	2.44×(53.4%)	4.57×
49	2.34×(51.2%)	2.23×(48.8%)	2.53×(55.4%)	2.37×(51.9%)	2.65×(58.0%)	2.60×(56.9%)	4.57×

Table 15 This table shows the performance of 1-Shot models using a subset of performance counters.

Machine-Compiler	Subset	Full Set	OPT
Nehalem-GCC	8.53×(92.2%)	8.79×(93.6%)	9.06×
Nehalem-ICC	4.51×(85.6%)	4.84×(83.6%)	5.60×
Q9650-GCC	3.24×(86.8%)	3.20×(87.9%)	3.67×
Q9650-ICC	3.69×(81.0%)	3.68×(84.4%)	4.57×

Table 16 This table shows the performance improvement of Multi-Models using the subset of performance counters found when using the LR models. Percentage values shown between parentheses indicates the average of the percentages of OPT (optimal over our optimization space) over all benchmarks

Analysis

Table 15 shows results of building our models using a subset of performance counters and used in a 1-shot scenario. For our 1-shot models, we observed that the LR models give the same performance for both the subset and full set of performance counters, except for Q9650-GCC. However, even this exceptional case shows only slight degradation, less than 0.05×, when compared to using the full set of counters. We found that the SVM, IBk, and MLP models trained with our performance counters subset

gave less improvements than using a full set. However, we noticed that some architecture/compiler configurations and machine learning algorithms did achieve improvements using the subsets of performance counters. For example, Nehalem-ICC with IBk achieved 60.7% of the optimization space optimal (OPT), while the same model trained with the full set of achieved only 53.8%. We also observed that we often achieve better average performance improvement with K* and M5P models with the subset of performance counters versus using the full set, especially for the K* model on the Nehalem-GCC configuration. For those models, we achieved 81.1% of OPT with the subset of performance counters, while we achieved 77% of OPT with the full set. Thus, we are able to build a model with better prediction quality using a smaller set of performance counters than using the full set.

Table 16 shows our results when using our multi-model models with the subset of performance counters. Using, the subset of counters slightly outperformed models versus using a full set for the Q9650 architecture. Although, the amount of improvements are not substantial, we can see the potential when using a subset of performance counters since we are able to build a model that achieves similar performance as when using the full set. In the case of the Nehalem configurations, we observed the opposite of the Nehalem results. Models that were trained with a full set of performance counters outperformed those models trained with a subset.

5.6 Evaluation of ML Algorithms Parameters

In this section, we discuss the impact of the various parameters of the machine learning algorithms we tested, and their relative impact on the quality of the predictors we build. Table 17 summarizes the results, in terms of average performance, for numerous different parameters values we have tested. We use the Weka option designations, and we mark with an asterisk the parameter configuration we have selected for the experiments presented in Section 5.2.

For LR, we evaluated different feature selection methods. $S = 0$ means we use the M5′ method, $S = 1$ means we do not use any feature selection method, and $S = 2$ means we use a greedy method. The default setting for Weka is $S = 0$. For all machine-compiler configurations except Q9650-GCC, we achieved the best prediction results with greedy method ($S = 2$). For Q9650-GCC, the prediction model with no feature selection gives the best prediction results by only a marginal fraction over the greedy method, and for three out of four configurations the greedy method provides the best results.

For SVM, we evaluated the *GaussianKernel* and the *NormalizedPolyKernel* kernel functions. We evaluated a range of gamma values (G) for *Gaussiankernel*, and a range of of exponent values (E) for *NormalizedPolyKernel*. For both kernels, we evaluated a different complexity values (C). We tried $G = 0$, 0.01(default), 25, 30, 50, 75, $C =$ 1(default), 2, 4, and $E = 1$, 2(default), 4, 8, 16. The best performance improvement is achieved with the *NormalizedPolyKernel* kernel, using $C = 1$ and $E = 8$ for all machine-compiler configurations.

For IBk, we tested with several number of neighbors $K = 1$(default), 2, 5. The default value for K is 1 in Weka, but we observe that using $K = 5$ gives the best

Algorithm and Parameter Configurations	Nehalem-GCC	Nehalem-ICC	Q9650-GCC	Q9650-ICC
LR -S 0	5.55	1.88	2.23	2.25
LR -S 1	4.77	1.96	**2.25**	2.06
LR -S 2 (*)	**5.69**	**2.13**	2.24	**2.34**
SVM NormalizedPolykernel -C 1.0 -E 8.0 (*)	**6.00**	**2.31**	**1.99**	**2.23**
SVM RBFKernel -C 2.0 -G 0.0	1.56	1.17	1.37	1.25
SVM RBFkernel -C 2.0 -G 25.0	4.19	2.21	1.65	1.30
SVM RBFKernel -C 2.0 -G 50.0	4.51	2.09	1.30	1.25
SVM RBFKernel -C 2.0 -G 75.0	4.11	2.07	1.25	1.25
SVM RBFKernel -C 4.0 -G 30.0	4.09	2.21	1.32	1.28
SVM RBFKernel -C 6.0 -G 30.0	3.89	2.21	1.42	1.28
SVM RBFKernel -C 0.01 -G 30.0	1.66	1.75	1.26	1.28
SVM RBFKernel -C 4.0 -G 50.0	3.92	2.07	1.30	1.25
IBk -K 1	6.22	3.01	**2.33**	2.53
IBk -K 2	6.07	2.88	2.13	**2.97**
IBk -K 5 (*)	**6.83**	**3.49**	2.32	2.94
M5P -M 1.0	6.17	2.99	2.21	2.67
M5P -M 2.0	6.17	2.99	2.21	**2.67**
M5P -M 4.0 (*)	**6.24**	**2.99**	**2.32**	2.65
M5P -M 10.0	5.29	2.83	2.23	2.05
M5P -M 50.0	6.24	2.66	1.89	2.62
K* -B 0 -M a	3.83	2.04	1.30	1.25
K* -B 20 -M a (*)	6.98	**3.42**	2.17	**2.37**
K* -B 25 -M a	6.98	3.39	**2.23**	2.11
K* -B 50 -M a	**7.24**	3.08	2.01	2.23
k* -B 75 -M a	6.57	3.06	1.98	2.22
K* -B 100 -M a	5.10	2.03	2.25	2.37
K* -B 0 -M n	3.83	2.04	1.30	1.25
K* -B 20 -M n	6.99	3.41	2.23	2.22
K* -B 25 -M n	6.98	3.39	2.23	2.11
K* -B 50 -M n	7.24	3.08	2.01	2.23
MLP -L 0.3 -N 500 -H a	5.26	3.19	2.09	2.60
MLP -L 0.05 -N 500 -H a	**6.02**	2.51	2.15	2.06
MLP -L 0.1 -N 500 -H a	5.17	3.18	2.24	**2.77**
MLP -L 0.5 -N 500 -H a	5.50	2.38	2.26	2.53
MLP -L 0.9 -N 500 -H a	3.93	2.54	2.18	2.41
MLP -L 0.4 -N 500 -H a	5.12	2.79	2.00	2.18
MLP -L 0.5 -N 1000 -H a	5.43	2.58	**2.25**	2.52
MLP -L 0.5 -N 1500 -H a	5.24	2.60	2.24	2.42
MLP -L 0.5 -N 500 -H t (*)	5.44	**3.24**	2.18	2.46

Table 17 Average Performance Improvement with Different Machine Learning Parameter Configurations for 1-Shot PC Model

prediction result for both compilers. Although this configuration does not give the best prediction for Q9650, the prediction result is very close to the best. Thus, we selected $K = 5$ for our experiments.

For M5P, we evaluated different values for M which indicates the minimum number of instances. We tried $M = 1, 2, 4$(default), 10, 50. $M = 4$ gives the best result for three machine-compiler configurations except Q9650-ICC, and $M = 2$ gives the best result for Q9650-ICC, but this is only $0.02\times$ higher than the performance improvement with $M = 4$. We selected $M = 4$, as the final parameter setting, which is also the default Weka setting.

For K* algorithm, we tuned B and M, where B is the parameter for global balancing, and M is the decision on how missing attribute values are handled. We evaluated $B = 0$, 20(default), 25, 50, 75, 100, and $M =$ a(default), t where each indicates a dif-

ferent way to handle missing values (a is to average column entropy curves, and t is to normalize over the attributes). $B = 20$, $M =$ a achieves the best prediction for both machines with ICC compiler, and is fairly close to the best for both machines with GCC.

For MLP, we tuned three different parameters: the learning rate (L), the number of iterations (N), and the number of hidden layers (H). We evaluated $L = 0.05$, 0.1, 0.3(default), 0.4, 0.5, 0.9, $N = 500$(default), 1000, 1500, and $H =$ a(default) and t. For H, a is defined as $(\text{number of attributes} + \text{number of classes})/2$, and t is defined as $(\text{number of attributes} + \text{number of classes})$. Each machine-compiler configuration requires different parameter tuning for the best performance improvement for MLP. We selected the configuration with L=0.5, N=500, H=t.

6 Related Work

In recent years, considerable research has been performed on iterative compilation, and its benefits have been reported in several publications [1, 17, 18, 24, 29, 33]. Iterative compilation has been shown to regularly outperform the most aggressive compilation settings of commercial compilers, and it has often reached performance comparable to hand-optimized library functions [25, 48, 56, 57].

Machine learning and search techniques applied to compilation has been studied in many recent projects [16, 34, 37, 51, 52, 60, 61]. These previous studies have developed machine learning-based algorithms to enabling efficiently search for the optimal selection of optimizing transformations, the best values for the transformation parameters, or the optimal sequences of compiler optimizations. Generally, these studies customize optimizations for each program or local code segments, some based on code characteristics.

For example, Monsifrot *et al.* [37] used decision trees to decide whether to enable or disable loop unrolling. This was one of the early efforts on using machine learning to tune a high-level transformation. They showed an improvement of 3% over a hand-tuned heuristic and 2.7% over g77's unrolling strategy on the IA64 and UltraSPARC, respectively. Stephenson *et al.* [52] used genetic programming to tune heuristic priority functions for three compiler optimizations within the Trimaran's IMPACT compiler. For one of the optimizations, register allocation, they were only able to achieve on average a 2% increase over the manually tuned heuristic. The results in these papers highlight the diminishing results obtained when only controlling a single optimization. In contrast, the research in this paper controlled numerous optimizations available in the PoCC compiler.

Kulkarni *et al.* [34] 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.* [17] 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, i.e., a genetic algorithm had to be *retrained* for each new program to decide the best optimization sequence for that program.

An innovative approach to iterative compilation was proposed by Parello *et al.* [41] where they used performance counters at each stage to propose new optimization sequences. An application was run and performance counter information was measured in order to identify performance *anomalies* that could be resolved by applying certain optimizations. The anomalies and proposed optimizations that could be applied to resolve them were encoded in a manually constructed decision tree. Even though this was a very systematic approach, the time required to manually construct this decision tree took weeks for each benchmark and was specific to a certain targeted architecture. In contrast, our technique does not need to generate performance counters during each iteration of optimizing the program, but instead a model is produced that can predict the best optimization sequences for a program. Also, we use machine learning to automatically construct the models used to predict the optimization configurations that were used.

Cavazos *et al.* [12] address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics. That work was limited to the traditional optimizations found in the PathScale compiler. Despite the numerous transformations considered, the complexity is not comparable to the restructuring transformations available in state-of-the-art polyhedral frameworks, such as the one we used in this work.

Park *et al.* [42] propose a novel program characterization technique, i.e., graph-based characterization, to use as input to a model that predicts optimizations to apply to a program. In this work, the authors characterize programs using the program's control flow graph (CFG), and they construct prediction models using SVMs with a shortest path graph kernel. These specialized graph kernels take as input characterizations that preserve the graph-based topology of the program, in contrast to previous characterization techniques that are represented as fixed-length vectors. The authors show that this method of characterizing programs is competitive with previous characterization techniques.

Chen *et al.* [13] developed the CHiLL infrastructure, a polyhedral loop transformation and code generation framework. Tiwari *et al.* [53] coupled the Active Harmony search engine to CHiLL to automatically tune some high-level transformation parameters, such as tile sizes. In this paper, we target quite a different search space, i.e., 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.

Bondhugula *et al.* [8, 9] proposed the first integrated heuristic for parallelization, fusion, and tiling in the polyhedral model 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 be implicitly tuned by minor ad-hoc modifications of Bondhugula's cost model. Pouchet *et al.* [45] performed empirical search to directly find the coefficients of the affine scheduling matrix in a polyhedral framework. While these results showed significant improvements on small kernels, the empirical search needed up to a thousand runs for larger benchmarks [44]. In this work, we have abstracted the scheduling matrix behind high-level polyhedral primitives and the as-

sociated cost models for selecting the enabling transformations, reducing the search space to as little as a few hundred possibilities in place of the billions of possible schedules. This enabled us achieve on average up to 80% of the optimization space optimal performance in no more than six runs.

7 Conclusion

The problem of improving performance of applications through compiler optimizations 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 compilers to each new 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 optimization combinations of loop transformations are needed to implement the most effective orchestration of tiling, parallelization, and vectorization. While all these optimizations have been studied independently, in practice, they must be evaluated together to achieve the best performance possible.

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.

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 run-time characteristics of a program (modeled with performance counters) with the speedup expected from a given polyhedral optimization configuration. We evaluated our approach using several machine learning algorithms, on a variety of benchmarks, and two different multi-core machines. For the test suite, the best points in our optimization search space yield an average $9\times$ speedup (with peaks of up to $28\times$) with GCC on an Intel Xeon E5620 and $3.6\times$ on Intel Xeon Q9650. Using the predictive machine learning models, testing at most six candidate optimization configurations on the target machine, we achieve an average speedup of $8.7\times$ on E5620 and $3.2\times$ on Q9650 with GCC as the backend compiler and an average speedup of $4.8\times$ and $3.6\times$ using Intel ICC as the backend compiler.

Acknowledgements This work was funded in part by the U.S. National Science Foundation through awards 0926688, 0811781, 0811457, 0926687 and 0926127, the Defense Advanced Research Projects

Agency through AFRL Contract FA8650-09-C-7915, the DARPA Computer Science Study Group (CSSG), the U.S. Department of Energy through award DE-FC02-06ER25755, and NSF Career award 0953667.

References

1. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO) (2006)
2. Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. International Journal of Machine Learning **6**, 37–66 (1991)
3. Almagor, L., Cooper, K., Grosul, A., Harvey, T., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 231–239. New York (2004)
4. Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammerling, S., Demmel, J., Bischof, C., Sorensen, D.: Lapack: a portable linear algebra library for high-performance computers. In: Proceedings of the 1990 ACM/IEEE conference on Supercomputing, Supercomputing '90, pp. 2–11. IEEE Computer Society Press, Los Alamitos, CA, USA (1990). URL http://dl.acm.org/citation.cfm?id=110382.110385
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (2004)
6. Baumgartner, G., Bernholdt, D., Cociorva, D., Harrison, R., Hirata, S., Lam, C.C., Nooijen, M., Pitzer, R., Ramanujam, J., Sadayappan, P.: A high-level approach to synthesis of high-performance codes for quantum chemistry. In: Supercomputing (2002)
7. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of the International Conference on Compiler Construction (ETAPS CC), LNCS 6011, pp. 283–303 (2010)
8. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Proceedings of the International Conference on Compiler Construction (ETAPS CC) (2008)
9. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI) (2008)
10. Bouckaert, R.R., Frank, E., Hall, M.A., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: WEKA–experiences with a java open-source project. Journal of Machine Learning Research **11**, 2533–2541 (2010)
11. Cavazos, J., Dubach, C., Agakov, F., Bonilla, E., O'Boyle, M.F., Fursin, G., Temam, O.: Automatic performance model construction for the fast software exploration of new hardware designs. In: International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES) (2006)
12. Cavazos, J., Fursin, G., Agakov, F.V., Bonilla, E.V., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO) (2007)
13. Chen, C., Chame, J., Hall, M.: CHiLL: A framework for composing high-level loop transformations. Tech. Rep. 08-897, U. of Southern California (2008)
14. Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., Wu, C.: Evaluating iterative optimization across 1000 datasets. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, PLDI '10, pp. 448–459. ACM, New York, NY, USA (2010). DOI 10.1145/1806596.1806647. URL http://doi.acm.org/10.1145/1806596.1806647
15. Cleary, J.G., Trigg, L.E.: K*: An instance-based learner using an entropic distance measure. In: In Proceedings of the 12th International Conference on Machine Learning, pp. 108–114. Morgan Kaufmann (1995)
16. Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: Acme: adaptive compilation made efficient. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 69–77. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1065910.1065921

17. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 1–9. ACM Press (1999)
18. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. Journal of Supercomputing **23**(1), 7–22 (2002)
19. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review **51**(1) (2009). DOI 10.1137/070693199. URL http://link.aip.org/link/?SIR/51/129/1
20. Dubach, C., Cavazos, J., Franke, B., O'Boyle, M., Fursin, G., Temam, O.: Fast compiler optimisation evaluation using code-feature based performance prediction. In: Proceedings of the International Conference on Computing Frontiers (CF) (2007)
21. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.: Portable compiler optimization across embedded programs and microarchitectures using machine learning. In: Proceedings of the International Symposium on Microarchitecture (MICRO) (2009)
22. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I: one dimensional time. International Journal of Parallel Programming (IJPP) **21**(5), 313–348 (1992)
23. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part II: multidimensional time. International Journal of Parallel Programming (IJPP) **21**(6), 389–420 (1992)
24. Franke, B., O'Boyle, M., Thomson, J., Fursin, G.: Probabilistic source-level optimisation of embedded programs. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 78–86. ACM Press (2005)
25. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2), 216–231 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation"
26. Fursin, G., Cavazos, J., Temam, O.: Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In: In Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC), pp. 245–260. Springer LNCS (2007)
27. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: MILEPOST GCC: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit (2008)
28. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations. International Journal of Parallel Programming (IJPP) **34**(3), 261–317 (2006)
29. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Automatic selection of compiler options using non-parametric inferential statistics. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 123–132 (2005)
30. INRIA, The Ohio State University: Polybench, the polyhedral benchmark suite. http://polybench.sourceforge.net
31. Irigoin, F., Triolet, R.: Supernode partitioning. In: ACM SIGPLAN Principles of Programming Languages, pp. 319–329 (1988)
32. Kelly, W., Pugh, W.: A unifying framework for iteration reordering transformations. In: IEEE Intl. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP'95), pp. 153–162 (1995)
33. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), p. 237 (2000)
34. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), pp. 171–182. ACM Press (2004)
35. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 201–214. ACM Press (1997)
36. Long, S., Fursin, G.: A heuristic search algorithm based on unified transformation framework. In: Proceedings of the International Conference on Parallel Processing Workshops (ICPPW), pp. 137–144 (2005)
37. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA), pp. 41–50. Springer-Verlag (2002)
38. Mucci, P.: Papi – the performance application programming interface. http://icl.cs.utk.edu/papi/index.html (2000)

39. Namolaru, M., Cohen, A., Fursin, G., Zaks, A., Freund, A.: Practical aggregation of semantical program properties for machine learning based optimization. In: International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES) (2010)
40. Orozco, D., Gao, G.R.: Mapping the FDTD Application to Many-Core Chip Architectures. In: ICPP (2009)
41. Parello, D., Temam, O., Cohen, A., Verdun, J.M.: Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In: Proceedings of the ACM/IEEE conference on Supercomputing (SC), p. 15. IEEE Computer Society (2004)
42. Park, E., Cavazos, J., Alvarez, M.A.: Using graph-based program characterization for predictive modeling. In: 10th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'12). IEEE Computer Society press, San Jose, CA (2012)
43. Park, E., Pouchet, L.N., Cavazos, J., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. In: 9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11), pp. 119–129. IEEE Computer Society press, Chamonix, France (2011)
44. Pouchet, L.N., Bastoul, C., Cohen, A., Cavazos, J.: Iterative optimization in the polyhedral model: Part II, multidimensional time. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), pp. 90–100. ACM Press (2008)
45. Pouchet, L.N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part I, one-dimensional time. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 144–156. IEEE Comp. Soc. press (2007)
46. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P.: Combined iterative and model-driven optimization in an automatic parallelization framework. In: Proceedings of the ACM/IEEE conference on Supercomputing (SC) (2010). 11 pages
47. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: Convexity, pruning and optimization. In: 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11), pp. 549–562. ACM Press, Austin, TX (2011)
48. Puschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: Spiral: Code generation for dsp transforms. Proceedings of the IEEE **93**(2), 232–275 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation"
49. Ramanujam, J., Sadayappan, P.: Tiling multidimensional iteration spaces for multicomputers. Journal of Parallel and Distributed Computing **16**(2), 108–230 (1992)
50. Smith, G.: Numerical Solution of Partial Differential Equations: Finite Difference Methods. Oxford University Press (2004)
51. Stephenson, M., Amarasinghe, S.: Predicting Unroll Factors Using Supervised Classification. In: CGO '05: Proceedings of the international symposium on Code generation and optimization, pp. 123–134. IEEE Computer Society, Washington, DC, USA (2005). DOI http://dx.doi.org/10.1109/CGO.2005.29
52. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.M.: Meta optimization: improving compiler heuristics with machine learning. SIGPLAN Not. **38**(5), 77–90 (2003). DOI http://doi.acm.org/10.1145/780822.781141
53. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–12. IEEE Computer Society (2009)
54. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT) (2009)
55. Voronenko, Y., de Mesmay, F., Püschel, M.: Computer generation of general size linear transform libraries. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 102–113 (2009)
56. Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical models for empirical search-based performance tuning. Int. J. High Perform. Comput. Appl. **18**(1), 65–94 (2004)
57. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the ACM/IEEE conference on Supercomputing (SC), pp. 1–27. IEEE Computer Society (1998)
58. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the atlas project. Parallel Computing (2000)
59. Wolf, M., Lam, M.: A data locality optimizing algorithm. In: ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation, pp. 30–44. New York (1991)

60. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI) (2003)
61. Yotov, K., Pingali, K., Stodghill, P.: Think globally, search locally. In: ICS '05: Proceedings of the 19th annual international conference on Supercomputing, pp. 141–150. ACM Press, New York, NY, USA (2005). DOI http://doi.acm.org/10.1145/1088149.1088168