Using Decision Tree Voting to Select a Polyhedral Model Loop Transformation

by

Ray Ruvinskiy

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Ray Ruvinskiy

Abstract

Algorithms in fields like image manipulation, sound and signal processing, and statistics frequently employ tight loops. These loops are computationally intensive and CPU-bound, making their performance highly dependent on efficient utilization of the CPU pipeline and memory bus. Recent years have seen CPU pipelines becoming more and more complicated, with features such as branch prediction and speculative execution. At the same time, clock speeds have stopped their prior exponential growth rate due to heat dissipation issues, and multiple cores have become prevalent. These developments have made it more difficult for developers to reason about how their code executes on the CPU, which in turn makes it difficult to write performant code. An automated method to take code and optimize it for most efficient execution would, therefore, be desirable. The Polyhedral Model allows the generation of alternative transformations for a loop nest that are semantically equivalent to the original. The transformations vary the degree of loop tiling, loop fusion, loop unrolling, parallelism, and vectorization. However, selecting the transformation that would most efficiently utilize the architecture remains challenging. Previous work utilizes regression models to select a transformation, using as features hardware performance counter values collected during a sample run of the program being optimized. Due to inaccuracies in the resulting regression model, the transformation selected by the model as the best transformation often yields unsatisfactory performance. As a result, previous work resorts to using a five-shot technique, which entails running the top five transformations suggested by the model and selecting the best one based on their actual runtime. However, for long-running benchmarks, five runs may be take an excessive amount of time. I present a variation on the previous approach which does not need to resort to the five-shot selection process to achieve performance comparable to the best five-shot results reported in previous work. With the transformations in the search space ranked in reverse runtime order, the transformation selected by my classifier is, on average, in the 86th percentile. There are several key contributing factors to the performance improvements attained by my method: formulating the problem as a classification problem rather than a regression problem, using static features in addition to dynamic performance counter features, performing feature selection, and using ensemble methods to boost the performance of the classifier. Decision trees are constructed from pairs of features (performance counters and structural features than can be determined statically from the source code). The trees are then evaluated according to the number of benchmarks for which they select a transformation that performs better than two baseline variants, the original program and the expected runtime if a randomly selected transformation were applied. The top 20 trees vote to select a final transformation.

Acknowledgments

I would like to thank my thesis supervisor, Peter van Beek, for the guidance, patience, and optimism he has offered over the past four years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, I informally introduce and motivate the problem that I address in this thesis. I also summarize the contributions of the thesis and the organization of the thesis.

1.1 The Problem

Loops are a fundamental part of many scientific computing algorithms, with applications in image manipulation, sound and signal processing, statistical simulations, and any other algorithms that rely on linear algebra, among others. Loops in such algorithms are usually *tight* — they are computationally intensive and run for many iterations without blocking to perform high-latency operations such as disk or network I/O. Accordingly, the performance of such loops is very dependent on efficient utilization of the CPU pipeline and the memory bus.

While in prior decades, software developers could rely on successive generations of new hardware to automatically to speed up tight loop execution due to ever-increasing CPU clock speeds, that is no longer the case. Recent years have seen a marked decline in the rate of CPU clock speed increases. Clock speeds of commodity Intel CPUs increased exponentially starting in the 1980's through 2005, from 4.77 MHz to 3 GHz. They have not, however, increased much since then. Top-of-the-line Intel processors at the time of writing (Haswell Core i7) top out at 3.5 GHz, while the processor with the highest clock speed is the mainframe IBM zEC12, at 5.5 GHz. The reason for the lack of clock speed increase is the difficulty CPU manufacturers have faced with effective heat dissipation at higher clock rates. Instead, CPUs have been designed with deeper and more complex pipelines, as well as an increasing number of cores. An effect of the more intricate CPU pipelines has been an increase in difficulty intuitively reasoning about how tight loops actually execute on the CPU. In addition, tight loops are by default compiled to single-threaded code, which is unable to take advantage of the multiple cores and hardware threads available on modern CPUs. Software developers must perform manual loop optimizations to ensure that the loops take advantage of the full range of capabilities provided by the CPU. Loops must be structured so as to effectively utilize instruction level parallelism, branch prediction, and

instruction and data prefetch capabilities. As well, in order to make use of multiple cores, software developers must manually parallelize the loops. Such manual transformations result in code that is difficult to read and to maintain. To complicate matters further, the transformations that would yield the most optimal execution time can vary with the original structure of the loops as well as the CPU model and vendor. To write optimal code, software developers must not only be domain experts in the field to which the algorithms they are developing belong, but they must also be CPU architecture experts.

Such cross-domain experts are difficult to find. Even in the case of developers skilled in CPU architecture, the need to consider the demands imposed by the CPU architecture complicate and confound the developers' job. It would, therefore, be desirable for it to be possible for domain-expert software developers to write straightforward loops tailored to the needs of the algorithm in question and have the necessary transformations and optimizations automatically applied at the compilation stage. To do so, the compiler must be able to accomplish two tasks:

1. Given a loop or a nest of loops, the compiler must be able to apply transformations that optimize the loops while preserving the semantics of the original code.

2. Out of the set of all possible correct transformations, or a subset thereof, the compiler must then be able to select a good transformation for the loop nest and the CPU architecture in question.

1.2 Contributions of the Thesis

My focus is on the polyhedral model, a mathematical framework wherein loops and loop nests can be represented with polyhedra [24]. The polyhedral model allows loop transformations to be expressed as algebraic operations on polyhedra and specifically allows the generation of a search space of loop transformations that preserve the semantics of the original loop. Many loop transformations are possible. I specifically examine several transformations commonly used when optimizing loops: loop tiling, loop unrolling, loop fusion, loop prevectorization, and loop parallelization. In previous work, various techniques have been proposed to choose a transformation to use for a candidate program from the transformation search space. Some early techniques employed an analytical model, while others relied in whole or in part on a search that required running programs generated from different transformations and comparing observed runtimes. Approaches that rely on a static model derived from the program structure to select a transformation do not take into account the architecture for which the code is compiled. As a result, the static models do not capture the tradeoffs imposed by the hardware. On the other hand, a search process that involves running multiple transformations can be very time consuming. Recently, Park et al. [23] used machine learning regression models to select transformations, using as features hardware performance counter values obtained while running the candidate program. Park et al. explored the use of linear regression and SVM regression to predict the speed-up of an arbitrary transformation from the search space over the original program.

The search space consisted of transformations that were composed of some combination of loop tiling, unrolling, fusion, prevectorization, and parallelization. The transformation predicted to have the largest speed-up was taken to be the best transformation. However, the lack of accuracy of the regression models resulted in the top transformation suggested by the model to have inadequate performance. To accommodate this shortcoming, Park et al. proposed a five-shot approach, where binaries generated by applying each of the top five transformations suggested by the model were run, and the transformation that had the best runtime was chosen. The five-shot approach, however, can be excessively an onerous one if the program being optimized is long-running. There has been evidence of developers being reluctant to use Profile-Guided Optimization techniques to extract more performance from a program due to the process being complicated and time consuming [12, p. 339]. It is, therefore, reasonable to assume a five-shot selection process would run into similar hurdles with developers.

I present a machine learning approach to learning a classifier that helps to determine which of a finite set of loop transformation sequences should be applied to a loop nest so as to minimize runtime. My approach achieves performance comparable to the best five-shot results reported by Park et al. without needing to resort to the five-shot selection process; it is sufficient to use the top transformation selected by the model. There are several key contributing factors to the performance improvements attained by my method: formulating the problem as a classification problem rather than a regression problem, using static features in addition to dynamic performance counter features, performing feature selection, and using ensemble methods to boost the performance of the classifier. Decision trees are constructed from pairs of features (performance counters and structural features than can be determined statically from the source code). The trees are then evaluated according to the number of benchmarks for which they select a transformation that performs better than two baseline variants: the original program and the expected runtime if a randomly selected transformation were applied. The top 20 trees vote to select a final transformation. On average, the performance of the transformations selected using my approach matches or exceeds the performance of the transformations selected with five-shot SVM regression, the highest-performing approach proposed by Park et al.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows.

In Chapter 2, I review background material necessary for the understanding of the thesis. I begin by describing the polyhedral model and how it can be used to generate correct loop nest transformations. I proceed to cover common loop transformation techniques and discuss their benefits and drawbacks. Finally, I conclude by introducing supervised machine learning for classification, describing the concept of features and classes, and cover one particular approach to classifier learning, decision trees.

In Chapter 3, I review relevant prior work on generating transformations using the polyhedral model and various approaches to choosing a good transformation. I also review

prior work on using machine learning to choose compiler optimizations in general and loop optimizations in particular.

In Chapter 4, I present a proposal for using a machine learning technique to choose a sequence of loop transformations among a finite search space of transformation sequences. I describe the full learning process, which includes data collection, feature selection, classifier construction, and application of the classifier.

In Chapter 5, I present an evaluation of my proposal. I demonstrate the efficacy of my technique by presenting a comparison of the runtime speed-ups obtained using my method with a baseline, as well as with results obtained by applying a prior state-of-the-art approach.

In Chapter 6, I summarize the results of the thesis and discuss some future work that could be undertaken.

Chapter 2

Background

In this chapter, I review the necessary background in the polyhedral model, loop transformations, and machine learning. For more background on these topics, see, for example, [26] and [30].

2.1 Polyhedral Model

The polyhedral model is a mathematical framework used to represent loops and facilitate loop transformations [26]. Under the polyhedral model, certain constructs in software source code can be represented as *polyhedra*. Algebraically, a polyhedron encompasses a set of points in a $\mathbb{Z}^n$ vector space satisfying the inequalities

$$\mathcal{D} = \{x \mid x \in \mathbb{Z}^n, Ax + a \geq 0\}, \tag{2.1}$$

where A is a matrix while x and a are column vectors. In the polyhedral model, A is a matrix of constants, x is a vector of iteration variables, a is a vector of constants [3], 0 is the zero vector, and the inequality operator ($\geq$) compares the vectors element-wise. Geometrically, a polyhedron may be thought of as a convex set of points in a lattice [2].

Polyhedra can be used to represent nested loops. Every statement executed as part of a loop iteration is referred to as an *execution instance*. In the polyhedral model, each execution instance that is part of a loop iteration is represented by a point on a polyhedron. The dependencies between the statements are indicated by directed edges, creating a directed acyclic graph [21].

The polyhedral model is limited in the nature of programming constructs it can represent. To be mapped to the polyhedral model, a block of statements must be restricted such that it is only enclosed in *if* statements and *for* loops. Pointer arithmetic, with the exception of array accesses, is disallowed. Function calls must be inlined. Loop bounds are restricted to affine functions of loop iterators and global variables. This means, for example, that loop bounds cannot be determined by a call to an arbitrary function and cannot be read at runtime. They must be statically and linearly derived from the loop iterator

and/or global variable values. The parts of a program that meet these requirements are referred to as *Static Control Parts* (SCoPs). While these limitations appear onerous at first glance, static control parts play a large role in scientific and signal processing kernels [15].

Loop bounds determine the *iteration domains* of the statements in consideration. An iteration domain represents the values taken on by the loop iterators for all iteration instances.

Example 2.1. *Given the source code*

```
1  for (i = 1; i <= n; i++)
2    for (j = 1; j <= m; j++)
3      if (i < j)
4        S(i, j)
```

the iteration domain $\mathcal{D}_S$ *of statement* S *in the polyhedral model becomes,*

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ m \\ -1 \end{pmatrix} \geq 0 \right\}.$$

The first two inequalities reflect the bounds imposed by the outer loop; the next two inequalities are the bounds imposed by the inner loop; and the last inequality reflects the logic of the conditional.

While iteration domains characterize all execution instances of each statement, they hold no information with respect to instance ordering. *Schedules* fill this gap. In the polyhedral model, a *timestamp* (or an *execution date*) is associated with each statement instance. The relative order of the timestamps reflects the execution order of the instances. A *scheduling function* maps execution instances to their corresponding timestamps [25]. The mapping of execution instances to timestamps is known as a *schedule*. In effect, the schedule is one variant of the program. If two instances share a timestamp, no partial ordering between them need be enforced, and they can therefore be executed in parallel.

Schedules may be *one-dimensional* or *multi-dimensional*. One-dimensional schedules have scalar timestamps. Multi-dimensional schedule timestamps are vectors expressed in big-endian order, with the first element the most significant and the last the least significant. For example, UNIX epoch timestamps can be used to make up a one-dimensional schedule, since they consist of a scalar integer representing the number of seconds since the beginning of the epoch. A timestamp of the form (year, month, day, hour, minute, second) is a multi-dimensional timestamp with a cardinality of 6 [7].

Since schedules are expressed as functions of loop iteration vectors, one-dimensional schedules can only be used to describe code blocks with single sequential loops, since the

instance ordering must in this case be a function of a single variable. Multi-dimensional schedules, on the other hand, allow for multiple nested sequential loops [26].

Schedules are affine. A statement S can have its schedule, θ_S, expressed as,

$$\theta_S(x_S) = T \begin{pmatrix} x_S \\ n \\ 1 \end{pmatrix},$$

where x_S is a loop iterator vector, n is a vector of global parameters, and T is a constant matrix containing coefficients that define the transformation.

Schedules allow us to express dependencies between instance executions. Two instances, R and S, are said to be in a *dependence relation* $\delta_{R,S}$ if both access the same memory address and one of the accesses is a write operation [26]. A polyhedral transformation is considered legal if it preserves all dependence relations within the polyhedron.

Example 2.2. *Consider the following source code:*

```
1  A[0] = 1;
2  for (i = 1; i <= n; i++) {
3    (R) A[i] = i;
4    (S) A[i − 1] = i ∗ A[i];
5  }
```

A legal one-dimensional schedule for statement R would be simply $\theta_R(i) = i$, and the schedule for statement S can be expressed as $\theta_S(i) = i + 1$. R and S are in a dependence relation for loop iterator values $i_S = i_R + 1$.

Dependence relations may also be expressed with *dependence polyhedra*. A dependence polyhedron $D_{R,S}$ is a subset of the Cartesian product of the iteration domains of statements R and S. Every dependence between two instances of the statements corresponds to a point on the polyhedron.

Example 2.3. *Consider once again the code in Example 2.2. The dependence polyhedron is defined by the inequality,*

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix} \begin{pmatrix} i_R \\ i_S \\ n \\ 1 \end{pmatrix} \geq 0.$$

The first line captures the relationship between i_R and i_S under which the dependence holds. The next two lines specify the bounds of i_R, while the last two lines specify the bounds of i_S.

Different schedules for statements also correspond to different transformations of the polyhedron. However, when performing optimizations, the transformations of interest are

only ones that are legal. Testing every schedule for legality quickly becomes impractical even with one-dimensional schedules, to say nothing of the multi-dimensional kind. It is, therefore, desirable to generate the set of legal schedules algorithmically.

A dependence relation $D_{R,S}$ implies that $\theta_R(x_R) < \theta_S(x_S)$. This inequality must be satisfied in all transformations. The affine form of Farkas' Lemma [9] is used in the process of deriving legal schedules.

Lemma 2.1 (Farkas' Lemma)**.** *Let D be a non-empty polyhedron defined by the inequalities $Ax + b \geq 0$. Then any affine function $f(x)$ is non-negative everywhere in D iff there exist λ_0 and λ such that,*

$$f(x) = \lambda_0 + \lambda^T(Ax + b),$$

where λ_0 is some non-negative scalar and λ is a row vector of non-negative scalars.

Using Farkas' Lemma, the schedule constraints may be expressed in an affine form. If $\theta_R(x_R) < \theta_S(x_S)$, then $\Delta_{R,S} = \theta_S(x_S) - \theta_R(x_R) - 1$ is non-negative at every point in the dependence polyhedron. By Farkas' Lemma,

$$\Delta_{R,S} = \lambda_0 + \lambda^T \left(D_{R,S} \begin{pmatrix} x_R \\ x_S \end{pmatrix} + d_{R,S} \right) \geq 0,$$

where $D_{R,S}$ is the dependence polyhedron for statements R and S and $d_{R,S}$ is a scalar component. All legal schedules satisfying the dependence relation also satisfy the inequality. Solving this system of inequalities, for instance using Fourier-Motzkin Elimination, allows these schedules to be enumerated [26].

In the multi-dimensional case [25], the scheduling constraint is extended to be,

$$\theta_R(x_R) \prec \theta_S(x_S),$$

where $\prec$ denotes a lexicographic ordering, with the elements of the scheduling vectors for the two instances being compared from the most to the least significant. For any of the time dimensions, the constaint can be either *weakly satisfied*, with $\theta_{R_i}(x_R) \leq \theta_{S_i}(x_S)$, or *strongly satisfied*, with $\theta_{R_i}(x_R) < \theta_{S_i}(x_S)$. For the overall dependence to hold, time dimension constraints can be weakly satisfied in lexicographic order until one of the time dimension constraints is finally strongly satisfied. Once one of the constraints is strongly satisfied, further time dimensions need no longer be examined.

It is necessary to select criteria to evaluate two schedules with respect to each other and to allow us to select the better one of the two. The *cost function* specifies these criteria. The most natural cost function for a particular machine would appear to be the measured execution time of each schedule. Computing the result of this cost function requires that each schedule being evaluated be rendered in source code form, compiled, and run on the target machine [25]. However, there are several drawbacks to this approach. The compilation and execution process may be time consuming and impractical for a large number of schedules. Program input, if applicable, must be carefully chosen to be representative if the nature of the input changes the execution path of the program. Care

must be taken when measuring the runtime, since the contents of the CPU cache when the program starts can affect the execution speed. Finally, this approach is less than ideal if the same binary is expected to be run on multiple machines, since without a theoretical model to guide schedule selection, nothing suggests how a schedule empirically found to be optimal for one architecture will perform on another architecture.

Alternatively, an analytic cost function or a static function derived from a performance model can be used. Finding an optimal schedule backed by a analytic model typically involves solving systems of affine inequalities, using Fourier-Motzkin Elimination or linear programming techniques. Early studies include models to minimize total execution time [10] and minimize synchronization [22]. More recently, Bondhugula et al. [4, 5] proposed a cost function suited to the parallelization of loops that selects schedules for minimal communication between running loop instances. After using tiling to extract coarse-grain parallelism from a loop nest, the cost function reflects the number of tiles traversed by an edge in the dependence polyhedron of a schedule.

2.2 Loop Transformations

Loop transformations are transformations to the structure of loop nests. The goal of the transformations is to speed up loop execution while preserving the original semantics of the loop nest. Transformations typically entail increasing or reducing the number of loops by, for example, increasing or reducing the depth of the nest while compensating for the change by altering the iteration domain, or by combining or splitting adjoining loops at the same depth.

2.2.1 Tiling

Loop tiling is a loop optimization technique that rearranges loop iteration domains so as to increase spatial locality and, potentially, expose parallelism [32, 31].

Consider the sample code in Listing 2.1, which represents the two-dimensional matrix multiplication operation $C = AB$.

Listing 2.1: Matrix Multiplication

```
1   int A[NI][NK];
2   int B[NK][NJ];
3   int C[NI][NJ];
4   /* populate A and B, zero out C */
5   for (int i = 0; i < NI; i++) {
6     for (int j = 0; j < NJ; j++) {
7       for (int k = 0; k < NK; ++k) {
8         C[i][j] += A[i][k] * B[k][j];
9       }
10    }
11  }
```

Line 8 of Listing 2.1 consists of two read operations, from A and B, and one write operation to C. Assuming a typical two-dimensional array memory layout that an int is 4 bytes in size, each iteration of the inner loop will access A with a stride of 4 bytes and B with a stride of 4·NJ bytes. Assuming a cache line size of 64 bytes, B[k][0], B[k][1],..., B[k][15] will be loaded into cache when B[i][0] is accessed. However, B[k][1],..., B[k][15] will have likely been evicted from the cache by the time they are accessed during subsequent iterations of the loop on line 6. As a result, a cache miss will be incurred on each of the subsequent accesses of B[k][1],..., B[k][15].

Suppose, however, that loop tiling is used and the loops are written as in Listing 2.2.

Listing 2.2: Tiled Matrix Multiplication

```
1   int A[NI][NK];
2   int B[NK][NJ];
3   int C[NI][NJ];
4   /* populate A and B, zero out C */
5   for (int i = 0; i < NI; i += 16) {
6     for (int j = 0; j < NJ; j += 16) {
7       for (int k = 0; k < NK; k += 16) {
8         for (int ii = i; i < min(ii + 16, NI); i++) {
9           for (int jj = j; j < min(jj + 16, NJ); j++) {
10            for (int kk = k; k < min(kk + 16, NK); k++) {
11              C[ii][jj] += A[ii][kk] * B[kk][jj];
12            }
13          }
14        }
15      }
16    }
17  }
```

The tile size used is 16. It was chosen since it is the quotient of the cache line size (64) and the data type size (4). The iteration space is then partitioned into rectangles, and the cache misses for B[kk][1],..., B[kk][15] are avoided, since the corresponding cache line will still be in cache when those memory addresses are accessed. The appropriate tile size depends on the particulars of the loop nest, such as the number of loops, the depth of the nest and the memory accesses, and the size of the array data type.

2.2.2 Fusion

Loop fusion, otherwise known as *loop jamming*, refers to the process of combining multiple loops into one. If done judiciously, benefits typically include increases in spatial locality and cache hit rates. In some cases, however, the effect may be the opposite [19].

Consider the sample code in Listing 2.3, extracted from the `gemver` benchmark (vector multiplication and matrix addition), which computes $x = Ay + z$.

Listing 2.3: gemver

```
1   int A[N][N];
2   int x[N];
3   int y[N];
4   int z[N];
5   /* populate A, y, z; zero out x */
6   for (int i = 0; i < N; i++) {
7     for (int j = 0; j < N; j++) {
8       x[i] = x[i] + A[j][i] * y[j];
9     }
10  }
11
12  for (int i = 0; i < N; i++) {
13    x[i] = x[i] + z[i];
14  }
```

Both the statements on line 8 and line 13 access x[i]. However, they reside in different outer loops. The element x[i] is initially loaded into cache when accessed in the first loop. Assuming an int size of 4 bytes and a cache line size of 64 bytes, x[0] will have likely been evicted from cache by the time it is accessed in the second loop if N > 15. Additionally, in the case that N > 31, all elements of x will have likely been evicted by the time they are accessed in the second loop. However, since the bounds of the two upper loops are the same, they can be fused, as show in Listing 2.4.

Listing 2.4: Fused gemver

```
1   int A[N][N];
2   int x[N];
3   int y[N];
4   int z[N];
5   /* populate A, y, z; zero out x */
6   for (int i = 0; i < N; i++) {
7     for (int j = 0; j < N; j++) {
8       x[i] = x[i] + A[j][i] * y[j];
9     }
10    x[i] = x[i] + z[i];
11  }
```

Fusing the loops in this instance makes it more likely that the second x[i] access occurs before x[i] has been evicted from cache and, as a result, that the memory access does not incur a cache miss.

On the other hand, consider the code in Listing 2.5, extracted from the `atax` benchmark (matrix transpose and vector multiplication). The code computes the expression $A^T Ax$.

Listing 2.5: atax

```
1   int A[NX][NY];
2   int x[NY];
3   int y[NY];
4   int tmp[NX];
5   /* Populate A, x, and y */
6   for (i = 0; i < NX; i++) {
7     tmp[i] = 0;
8     for (j = 0; j < NY; j++) {
9       tmp[i] = tmp[i] + A[i][j] * x[j];
10    }
11    for (j = 0; j < NY; j++) {
12      y[j] = y[j] + A[i][j] * tmp[i];
13    }
14  }
```

The inner loops on line 8 and line 11 have the same bounds. Also, the data dependencies involving tmp[i] between the two statements in the loops are such that they can therefore be fused, as in Listing 2.6.

Listing 2.6: atax, fused

```
1   int A[NX][NY];
2   int x[NY];
3   int y[NY];
4   int tmp[NX];
5   /* Populate A, x, and y */
6   for (i = 0; i < NX; i++) {
7     tmp[i] = 0;
8     for (j = 0; j < NY; j++) {
9       tmp[i] = tmp[i] + A[i][j] * x[j];
10      y[j] = y[j] + A[i][j] * tmp[i];
11    }
12  }
```

Indeed, it would appear that fusion would help with cache misses owing to tmp[i] accesses. However, that comes at the expense of an additional write operation in the same loop to a different location. This may introduce cache thrashing and interfere with the prefetcher. As a result, fusion in this instance may have an adverse effect on performance.

2.2.3 Unrolling

Iterations of tight loops often have computational operations that are independent of the execution result of a previous iteration of the loop. It should, therefore, be possible to schedule instructions pertaining to these operations in parallel, taking advantage of the instruction level parallelism functionality of modern CPUs. Also, loop counter variable

modification involves memory read and write operations, which may also incur a substantive cost relative to the body of the loop, depending on the work the loop performs. Reducing the number of loop counter variable manipulations can therefore yield noticeable runtime improvements.

Loop unrolling, also known as *loop unwinding*, is an optimization that repeats the body of the loop multiple times and offloads into the body of the loop logic typically handled in the loop termination condition and in the loop counter modification operation. The number of times that the body of the loop is repeated is referred to as the *loop unroll factor*. Repeating the body of the loop exposes instruction level parallelism and can yield substantial speedups [12, p. 354]. Depending on the contents of the body of the loop, similar savings can be achieved with respect to the loop counter manipulation costs.

Consider the code in Listing 2.7, extracted from the `jacobi-1d-imper` benchmark (1-D Jacobi stencil computation).

Listing 2.7: jacobi-1d-imper

```
1   int A[N];
2   int B[N];
3   /* Populate A */
4   for (int t = 0; t < TSTEPS; t++) {
5     for (int i = 1; i < N - 1; i++) {
6       B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
7     }
8     for (int j = 1; j < N - 1; j++) {
9       A[j] = B[j];
10    }
11  }
```

Applying a loop unroll factor of 4 to both inner loops, we get the code in Listing 2.8. For simplicity, the code in Listing 2.8 assumes that the arrays A and B have a size that is a multiple of 4, but it is possible to account for arrays of all sizes by treating the last k iterations, $k < 4$, separately.

Listing 2.8: jacobi-1d-imper, unrolled

```
1   int A[N];
2   int B[N];
3   /* Populate A */
4   for (int t = 0; t < TSTEPS; t++) {
5     for (int i = 1; i < N - 1; i += 4) {
6       B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
7       B[i + 1] = 0.33333 * (A[i-1 + 1] + A[i + 1] + A[i + 1 + 1]);
8       B[i + 2] = 0.33333 * (A[i-1 + 2] + A[i + 2] + A[i + 1 + 2]);
9       B[i + 3] = 0.33333 * (A[i-1 + 3] + A[i + 3] + A[i + 1 + 3]);
10    }
11    for (int j = 1; j < N - 1; j += 4) {
12      A[j] = B[j];
13      A[j + 1] = B[j + 1];
```

```
14          A[j + 2] = B[j + 2];
15          A[j + 3] = B[j + 3];
16        }
17      }
```

In Listing 2.8, the loop counter variables are directly used as array indices. All array indices in the bodies of the loops are, therefore, a constant offset from the loop counter variable value. As a result, unrolling the loop enables the compiler to directly precompute all memory location accessed, allowing us to fully realize the benefits from the loop unrolling optimization. That may not necessarily be possible if the loop counter variables are, for example, used as arguments for another function call, the return value of which is then used to index arrays, as in Listing 2.9.

Listing 2.9: Indirect array indexing

```
1   int function(int argument);
2
3   int A[N];
4   int B[N];
5   /* Populate A */
6   for (int t = 0; t < TSTEPS; t++) {
7     for (int i = 1; i < N - 1; i++) {
8       B[i] = 0.33333 * (A[i-1] + A[function(i)] + A[i + 1]);
9     }
10  }
```

2.2.4 Vectorization

Some CPU architectures provide support for *vectorized* or *Single Instruction Multiple Data* (SIMD) instructions. Such an instruction can operate on multiple memory locations at once. Common modern implementations include Intel's SSE2 extensions and the AltiVec instruction set in POWER and PowerPC CPUs. Vector instructions can often speed up tight array-processing loop execution by processing multiple array elements simultaneously. However, the compiler will typically only generate vectorized code for a loop if it determines the loop is suitable for the transformation. The use of vector instructions can impose certain requirements on the code. For instance, SSE2 instructions require that memory locations be aligned to 16-byte boundaries. Further, the compiler may have a higher chance to vectorize a loop if the body of the loop operates on data that corresponds to the instruction vector size in length. Accordingly, it may be beneficial to tile a loop with a tile size that equals the vector size [20].

Consider the code in Listing 2.10, extracted from the `lu` benchmark (LU decomposition).

Listing 2.10: lu

```
1   int A[N];
2   /* Populate A */
3   for (int k = 0; k < N; k++) {
4     for (int j = k + 1; j < N; j++) {
5       A[k][j] = A[k][j] / A[k][k];
6     }
7     for(int i = k + 1; i < N; i++) {
8       for (int j = k + 1; j < N; j++) {
9         A[i][j] = A[i][j] - A[i][k] * A[k][j];
10      }
11    }
12  }
```

Assuming instructions can operate on vectors of size 4, i.e., four memory locations at a time, the code in Listing 2.10 can be transformed into the code in Listing 2.11, which operates on arrays of size 4. Such array operations can then be easily implemented using vector instructions.

Listing 2.11: lu, prevectorized

```
1   int A[N];
2   int temp[4];
3   int temp2[4];
4   /* Populate A */
5   for (int k = 0; k < N; k++) {
6     for (int j = k + 1; j < N; j += 4) {
7       for (int ti = 0; ti < 4; ti++)
8         temp[ti] = A[k][j + ti];
9       for (int ti = 0; ti < 4; ti++)
10        temp[ti] = temp[ti] / A[k][k];
11      for (int ti = 0; ti < 4; ti++)
12        A[k][j + ti] = temp[ti];
13    }
14    for(int i = k + 1; i < N; i++) {
15      for (int j = k + 1; j < N; j += 4) {
16        for (int ti = 0; ti < 4; ti++)
17          temp[ti] = A[i][j + ti];
18        for (int ti = 0; ti < 4; ti++)
19          temp2[ti] = A[k][j + ti];
20        for (int ti = 0; ti < 4; ti++)
21          temp[ti] = temp[ti] - A[i][k] * temp2[ti];
22        for (int ti = 0; ti < 4; ti++)
23          A[i][j + ti] = temp[ti];
24      }
25    }
26  }
```

2.3 Machine Learning

As seen previously, numerous transformations can be applied to a loop nest while satisfying the dependencies of the execution and preserving correctness. These transformations can have a beneficial or an adverse effect on the runtime of the program. Different transformations can affect different programs differently, and transformation parameters may also have a substantial effect on performance. The transformations may also interact with each other and cannot be selected independently. Selecting a tile size, choosing which loops to fuse, selecting a loop unrolling factor, or determining which loops benefit from fusion is a non-trivial task even for machine architecture experts. Predicting how code structure maps onto and interacts with the architecture of the hardware on which it is running can be challenging and unintuitive. Being able to select the transformations and their parameters in an automated way can, therefore, be advantageous.

There are two main approaches to deriving algorithms to automatically choose the appopriate transformation or sequence of transformations. The first approach is manual, where a domain expert in machine architecture and compiler optimization leverages his or her experience to construct a set of rules that are applied when optimizing a program. The second approach is more automated, with the rules that govern which transformations are to be applied being generated with limited to no human intervention. *Machine learning* can be used as part of the automated approach. Machine learning, as the term will be used here, detects patterns in existing data sets and converts these patterns into rules. These rules can then be applied to new, previously unseen, data to obtain predictions (see [30]). The input to the machine learning algorithm is referred to as the *training set*. The unseen data used to gauge the effectiveness of the rules generated by the machine learning algorithm is called the *test set*.

Machine learning can be *unsupervised* or *supervised*. The goal of unsupervised machine learning is to detect patterns and derive rules for unlabelled data. In other words, an unsupervised machine learning algorithm is not provided with data to evaluate its predictive performance. In contrast, the input to a supervised machine learning algorithm is labelled, allowing result verification. Supervised algorithms generally provide more accurate rules than unsupervised ones. Since labelled training data for the problem covered by this thesis can be easily generated, I use a supervised algorithm.

2.3.1 Features and Classes

To illustrate how supervised machine learning works, I will use a running example commonly used in the literature: the weather problem. The weather problem involves making a decision as to whether some unnamed game should be played based on consideration of several weather conditions. The training set, or data set, for the weather problem is laid out in Table 2.1 (see [30]). In this data set, `outlook`, `temperature`, `humidity`, and `wind` are the *features* or *attributes*, and `play` is the decision, or the *classification*, reached based on the values of the features. Given this data set, we would like to be able to reach a decision on whether to play given a new and previously unseen set of feature values, for

example as presented in Table 2.2. The table already contains the decision for a similar set of attributes, with only the `temperature` value being different. Does that change the decision of whether to play?

Table 2.1: Weather Data Set

outlook	temperature	humidity	wind	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

Table 2.2: Weather Test Data Set

outlook	temperature	humidity	wind	play
overcast	hot	high	true	no

The most straightforward, and naive, way to derive a set of rules from the data set in Table 2.1 is to derive a rule from each row in the table. The rules corresponding to the first five rows are in algorithm 2.1.

Algorithm 2.1: Per-row weather rules

if *outlook is sunny and temperature is hot and humidity is high and there is wind* **then** no play;
else if *outlook is sunny and temperature is hot and humidity is high and there is no wind* **then** no play;
else if *outlook is overcast and temperature is hot and humidity is high and there is no wind* **then** play;
else if *outlook is rainy and temperature is mild and humidity is high and there is no wind* **then** play;
else if *outlook is rainy and temperature is cool and humidity is normal and there is no wind* **then** play;
...

However, this leaves us with rules that would not be able to handle data not previously seen. We would, therefore, not be able to make predictions for any feature combination not already specified in our training set table. Alternatively, we could derive the rules in algorithm 2.2 (see [30]).

Algorithm 2.2: Weather rules

if *outlook is sunny and humidity is high* **then** no play;
else if *outlook is rainy and there is wind* **then** no play;
else if *outlook is overcast* **then** play;
else if *humidity is normal* **then** play;
else play;

Applying this sequence of rules would produce the correct answer for every row in Table 2.1. Without having additional data, however, the rules cannot be tested, and it is impossible to tell if they are entirely reflective of reality. For example, if there were more data containing the row in Table 2.2, the rules would produce the wrong answer for that feature combination, as overcast outlook yields a prediction of `play`. The problem of generating rules that hew very closely to the training data but are not general enough is referred to as *overfitting*. In order to effectively learn, the training data should be as representative as possible, and the features used as input to the learning algorithm should correlate with the class that is being learned in general, rather than only for the training data. For example, in an extreme case, it is possible to sequentially number all rows in the training data and use the sequence number as one of the features. The machine learning algorithm could generate rules to predict the class from the sequence number with complete accuracy. However, such rules would be completely useless on anything other than the particular ordering of the training data elements that was used.

The features in Table 2.1 are all *symbolic* or *nominal*, in other words, the values are members of a fixed enumerated set. The value of `outlook` is always one of `sunny`, `overcast`, or `rainy`; the value of `temperature` is always one of `hot`, `mild`, or `cool`; et cetera. However, features could also be *numeric* or *continuous*. The weather data set could have a numeric feature if, for example, the temperature was instead expressed in degrees. As will be discussed in greater detail in later sections, continuous features pose a greater challenge to machine learning techniques.

Similarly, the class in Table 2.1 is also symbolic. In fact, it is a special case of a symbolic class: a *boolean* class, with only two possible values. Boolean classes are easier to predict (see [30]). However, a class could potentially also be numeric. For instance, the same set of features in Table 2.1 could be used to predict the number of people expected to turn out to an event.

2.3.2 Machine Learning Evaluation

Evaluating the efficacy of a machine learning approach, therefore, requires evaluating the rules it produces on previously unseen data. Certain problems lend themselves easily to this kind of evaluation, network traffic classification being one example. Suppose that a

network administrator is interested in obtaining information on the breakdown of network traffic by protocol. The network administrator can capture sample traffic from the network, manually classify it, and use it as the input, or training data, for a machine learning algorithm. Depending on how well the features are chosen and the effectiveness of the algorithm itself, the output produced by the algorithm may or may not be useful. The extent to which it is useful can be tested by having the rules generated by the machine learning algorithm classify new network traffic and seeing if the classification is correct. Since new network traffic is continuously generated, there is presumably no shortage of test data.

In other situations, however, new test data may be more difficult to generate. This usually happens when it is time consuming or otherwise expensive to generate samples. For example, consider the hypothetical case of scientists wishing to predict something with respect to the behaviour of periodical cicadas once they appear above ground. Periodical cicadas develop in 13 or 17-year cycles, depending on the brood. The development of all members of the brood is synchronized. The cicadas reside below ground for the entire length of their development cycle in their nymph (juvenile) form. At the end of the cycle, all members of the brood emerge above ground together, where they spend several weeks. During that time period, the now-adult cicadas mate, lay eggs, and die off. Nymphs proceed to hatch out of the eggs, whereupon they burrow into the ground, to reemerge 13 or 17 years later. If scientists collect data about the cicadas and apply a machine learning algorithm to that data, they will not be able to collect new test data for another 13 to 17 years, an iteration cycle almost certainly considered far too long.

Several approaches can be taken when data is in short supply, the common denominator between them being that some of the data available is used for training, while the rest is held back and used for testing. This division of data is used only for evaluation purposes, to generate error estimates. When generating the production-use classifier, all available data is used for training.

N-Fold Cross Validation

The *n-fold cross validation* technique requires partitioning the data into n parts, known as *folds*. The assignment of different data instances to folds is done randomly, although care is taken to ensure an even class distribution among all folds. The machine learning algorithm is then run n times. Each of the times, a different fold is used for the test data, while the remaining $n-1$ folds are used for the training data. An average of the error rates of all the runs is then taken. A value of $n = 10$ has been found to be yield good results in practice (see [30]), so 10-fold cross-validation is commonly used.

Leave One Out

The *leave one out* technique is a special case of n-fold cross validation and is useful in cases where the data available lends itself to natural, non-random partitioning. An example of this is data pertaining to computer program analysis, which is discussed at greater length

in later chapters. If data pertaining to n computer programs is available, each computer program can then become a fold. The machine learning algorithm is then run n times, each time using the data pertaining to a different computer program for testing and the data pertaining to the other $n - 1$ programs as training data. Again, an average of the error rates of all the runs is taken as the final estimate.

2.3.3 Supervised Machine Learning Algorithms

Supervised machine learning can be broadly categorized as either *classification* algorithms or *regression* algorithms. The output of a classification algorithm is one of a small number of symbolic classes, while the output of a regression algorithm is a real number. Of the many algorithms available, I review decision trees for classification, linear regression, and support vector machines for regression. Decision trees are reviewed at length, as they are the primary machine learning algorithm used in this thesis. Linear regression and support vector machines for regression are mentioned, as they form the basis for the immediate previous work upon which this thesis builds.

Decision Trees

Decision trees are the machine learning algorithm used in the approach presented in this thesis. Intermediate nodes in a decision tree correspond to features to be tested, while leaf nodes correspond to the decision rendered by the tree, or the classification. A data instance is evaluated by testing its feature values. Starting at the root node, the value of the feature determines which child node should be evaluated next. This is referred to as *branching on an attribute*. The process proceeds recursively until a decision is reached.

For symbolic features, the number of child nodes equals the number of possible values for that feature. In such a case, only one node in the tree corresponds to the feature, since the feature need only be evaluated once. Numeric features, however, may need to be evaluated or branched on several times. In the case of numeric features, the test typically involves checking if the value of the feature falls above or below a threshold value. In such a case, a numeric feature node will have two children. Alternatively, a numeric feature may be evaluated against multiple thresholds, resulting in multiple child nodes. A numeric feature may therefore be branched on several times along a path to a leave node. For example, at one level one subtree may correspond to a feature value less than 30 while the other subtree would correspond to a feature value greater than or equal to 30. Further down the left subtree, taken if the value is less than 30, the feature may be evaluated again. This time, the two subtrees may correspond to the feature values being less than 5 or greater than or equal to 5 (but still less than 30). As a result, numeric features result in deeper and more complicated trees, as well as potential overfitting to the training data. A coping strategy may be to *discretize* or *bin* the feature values, rendering the feature symbolic.

When constructing a decision tree, the feature evaluated at each level is chosen by evaluating the information gain from branching on each of the features available at that

level. All else being equal, the construction algorithm prefers shallower and simpler trees to deeper and more convoluted ones. Accordingly, it attempts to minimize the number of feature splits necessary to get to a leaf node.

To obtain the potential information gain from every available feature, the information value of the attribute values is calculated. The information value corresponds to the additional amount of information required at this stage to unambiguously classify the instance. It stands to reason that the more even the classification split between the different values of the feature, the more ambiguous the classification, the higher the information value, and the lower the information gain. The information value of an intermediate node can be subtracted from the information value of the parent node to obtain the *information gain* of the intermediate node. The higher the information gain, the more the branch from the parent node to the intermediate node contributes to the classification. The *entropy* of the distribution of feature values is used to represent the information value.

The equation to obtain the entropy is,

$$\text{entropy}(p_1, p_2, \ldots, p_n) = -p_1 \log p_1 - p_2 \log p_2 - \ldots - p_n \log p_n, \tag{2.2}$$

where n is the number of classes and $p_1 + p_2 + \ldots + p_n = 1$. The p_i values are derived from the split in classifications arising from each attribute value.

For example, returning to the weather data set in Table 2.1, there are 14 data instances in total, 9 of which are classified as `yes` and 5 as `no`. The overall information value is therefore,

$$\text{entropy}\left(\frac{9}{14}, \frac{5}{14}\right) = -\frac{9}{14}\log\frac{9}{14} - \frac{5}{14}\log\frac{5}{14} = 0.940$$

Considering the `outlook` attribute, it has three possible values: `sunny`, which yields 2 `yes` instances and 3 `no` instances, `overcast`, which yields 4 `yes` instances and 0 `no` instances, and `rainy`, which yields 3 `yes` instances and 2 `no` instances. The overall entropy of the attribute is therefore,

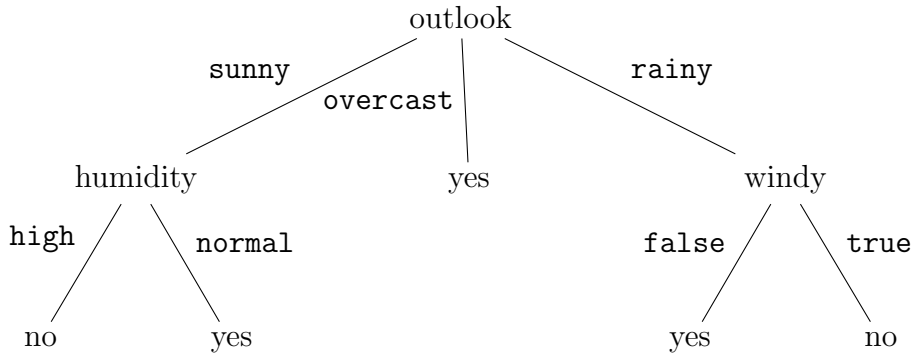
$$\text{entropy}\left(\frac{3}{5}, \frac{2}{5}\right) + \text{entropy}\left(\frac{4}{4}, \frac{0}{4}\right) + \text{entropy}\left(\frac{2}{5}, \frac{3}{5}\right) = 0.693$$

The information gain for branching on `outlook` is, therefore, $0.940 - 0.693 = 0.247$. Similar calculations show that the information gain from branching on `temperature` would be 0.0029, from branching on `humidity` would be 0.152, and from branching on `wind` would be 0.048. As branching on `outlook` yields the largest information gain, that attribute is chosen to branch on at the root of the tree. The information gain calculations are then performed at each intermediate node of the tree to determine the next attribute to split on. The final decision tree for the weather problem is depicted in Figure 2.1.

Regression Techniques

While classification techniques are used to predict discrete classes, real-valued labels are predicted using regression techniques. Regression techniques involve the study of the re-

Figure 2.1: Decision tree for the weather problem specified in Table 2.1.



lationship between one or more *independent variables* and one *dependent variable*. In machine learning, the dependent variable is the label, while the independent variables are the features. In this section, I will briefly discuss to regression techniques: linear regression and SVM regression.

Linear regression is used for predicting real-valued classes given a set of numeric features (see [30]). In linear regression, the output, x, is expressed as a linear combination of the feature values, $a_1, \ldots, a_n$, with the coefficients, $w_0, \ldots, w_n$, derived from the training data set:

$$x = w_0 + w_1 a_1 + \ldots + w_n a_n \tag{2.3}$$

The core of a linear regression approach is to choose $w_0, \ldots, w_n$ such that the sum of the squares of the distances between the predicted and actual classes across all training data instances is minimized. The main shortcoming in the linear regression approach is evident in its name. If the relationship between the features values and the class is not linear, it will be difficult to impossible to choose the coefficients in such a way that the predicted class values closely track the actual values.

Support vector machines, abbreviated as SVM, combine linear and non-linear terms, allowing for linear segments to be combined in such a way as to create overall non-linear functions (see [30]). The support vectors are instances selected from each class, with the selection performed in such a way as to maximize the distance between the support vectors of the different classes. SVM was originally introduced as a binary classification technique, but it has since been extended to apply to multi-class classification and regression problems.

2.3.4 Ensembles of Classifiers

When dealing with heterogeneous and diverse data, different features may be better suited to classifying different subsets of the data. It is, therefore, tempting to use feature vectors with high dimensionality that encompass all features that improve the classifier for some input. However, a high feature vector dimensionality carries a cost. It complicates the model and may impede learning, as well as requiring larger training samples. It can,

therefore, be beneficial to generate multiple classifiers (or *base-learners* from smaller feature sets) and combine their results [1, pp. 419-421].

The simplest way to combine the outputs of multiple classifiers is *voting*, with the final class being obtained by,

$$\sum_{j=1}^{L} w_j \cdot d_{ji},$$

where L is the number of base learner classifiers, d_{ji} is the vote of base learner classifier j for class C_i, and w_j is the weight assigned to base learner classifier j. The simplest and most widely-used voting technique is *simple voting*, where all classifiers are equally weighted, in other words, $\forall j : w_j = \frac{1}{L}$. The combination function for simple voting is the sum function. Other, more complex, combination functions include weighted sum, median, minimum, maximum, and product [1, pp. 424-425].

2.4 Summary

In this chapter, I provided an overview of the polyhedral model of loops and data dependencies, of loop transformations, and of machine learning. I discussed the construction of dependence polyhedra for static control parts, as well as the algorithmic generation of one-dimensional and multi-dimensional schedules for static control part statements. I described four different loop optimization techniques: tiling, fusion, unrolling, and vectorization. Finally, I introduced supervised machine learning, with an emphasis on decision trees.

Chapter 3

Related Work

In this chapter, I review prior work pertaining to using the polyhedral model to optimize loop execution, as well as work on using machine learning for general runtime and loop efficiency optimizations.

3.1 Optimizations in the Polyhedral Model

While the polyhedral model allows for the easy generation of a search space of semantically correct transformations, or schedules, multiple classes of approaches have been proposed for choosing or finding a good schedule. Some approaches use an analytic model to determine the best schedule to use, others rely on an iterative search that requires the compilation and running of successive schedules before a decision is made, while others still combine the two approaches.

Feautrier [11] presents two ways of choosing a schedule. Valid schedules can be discovered by solving the system of affine inequalities describing the dependencies between the execution instances. The system of inequalities can be solved using Fourier-Motzkin Elimination. However, Fourier-Motzkin Elimination has super-exponential complexity and quickly becomes infeasible as the number of dependencies increases. An alternative is a variation of the Integer Programming Simplex method called Parametric Integer Programming [8]. While this method is also exponential in the worst case, empirical results suggests that it is nonetheless faster than Fourier-Motzkin Elimination. Feautrier finds that the optimal schedule depends on the dimension being optimized. As an example, a schedule minimizing the latency can be obtained using linear programming, since latency can be expressed as a linear function of schedule coefficients.

Lim and Lam [22] propose an algorithm to select a transformation that maximizes parallelism and minimizes synchronization. The algorithm applies to programs with arbitrary loop nesting levels and loop sequences and consists of three phases. The first phase attempts to find a partitioning of statements that would yield full parallelism with no need for synchronization. The second phase attempts to find a schedule where the number of synchronization points is constant with respect to the number of loop iterations. Finally,

the third phase examines schedules with synchronization points linear in the number of iterations. Coarse-level parallelism is preferred to fine-grained parallelism provided the degree of parallelism (or the fanout) is the same in both cases and coarse-level parallelism reduces the need for synchronization. As the algorithm employs techniques such as Fourier-Motzkin Elimination, it has super-exponential complexity in the worst case. Also, while the algorithm minimizes the order of synchronization, it does not take into account the volume of communication [4].

Pouchet et al. [26, 25, 24] introduce an approach to find an optimal schedule using iterative compilation for single-dimensional and multi-dimensional schedules. A potentially large search space of valid schedules for a given kernel is constructed and subsequently traversed. Rather than using a theoretical model, the effectiveness of a schedule is judged by generating the corresponding source code, compiling it, and running it on the target machine.

Multiple approaches to traversing the search space were investigated:

1. Exhaustive search, where every valid schedule is run. This approach is obviously infeasible for all but the smallest program kernels.

2. A decoupling heuristic. As noted above, schedules are three-dimensional functions of iterator coefficients, parameter coefficients, and constant coefficients. The decoupling heuristic optimizes each dimension separately. The optimal iterator coefficients are selected first, while choosing the other coefficients at random and holding them constant. Then, given the optimal iterator coefficients, the optimal parameter coefficients are selected. Finally, with the first two dimensions fixed, the constant coefficients are chosen.

3. A genetic algorithm, where every step in the search space involves a mutation of the schedule. The mutation has to preserve the validity of the schedule. A probability distribution is calculated, and it is used to decide which coefficient to alter first. The new value of the coefficient is randomly chosen from within the range of valid values. If the change to the coefficient renders the schedule invalid, the other coefficients are adjusted to construct a valid schedule.

Bondhugula et al. [4, 5] propose an algorithm that generates schedules which minimize communication among execution instances. The algorithm applies to arbitrarily-nested loops. The approach is fully automated and requires no manual intervention. Coarse-grained parallelism is extracted by means of *loop tiling*, which involves partitioning the iteration domain of a loop into tiles (or blocks) which can be executed in parallel, with communication between threads necessary only before and after a block executes. Another advantage to tiling is that it improves data locality and hence cache utilization, since the smaller sizes of the tiles reduces cache line contention. The cost function to be minimized represents the number of tiles traversed by an edge in the dependence polyhedron. The smaller the quantity, the less inter-tile communication is necessary during execution. While the cost function is non-linear, it can bounded by an affine function using an application of Farkas' Lemma, and Integer Linear Programming can be used to find an optimal solution.

For every statement, a number of linearly independent solutions equal to the size of the iteration domain of the statement is required. Empirical tests show a 1.1–5.7× speed-up over state-of-the-art on a single-core machine and 1.5–7× speed-up on a multi-core machine. The algorithm does not take into account the characteristics of target hardware and relies solely on a theoretical cost model.

Pouchet et al. [27, 28] combine the approaches laid out in [25] and [4]. The approach is a combination of the use of analytic models, where they are available and effective, and empirical search for cases where analytic models do not account for important properties of the hardware which have a significant impact on performance. Specifically, the tiling approach Bondhugula et al. [4] is used, while also trying to maximize SIMD (single instruction, multiple data) parallelism via loop vectorization. An observation made in the work is that loop fusion and distribution drive the success of other transformations (vectorization, tiling, array construction). The authors also demonstrate, using an example, that different schedules may be optimal for different CPU chips (Xeon and AMD in the example, with a runtime difference of 25%). Thus, while an analytic model is used to select tiling and vectorization optimizations, a compile-and-run iterative process is used to select the program partitioning. The partitioning is selected first, followed by the application of tiling and vectorization optimizations. Vectorization is accomplished by computing the maximum distance between any two consecutive memory accesses in a loop and shifting loops having the smallest such distance inwards, towards the innermost position in the loop nest. Empirical results showed a performance improvement of $2 - 2.5\times$ over that of the compiler.

The iterative approaches described in this section suffer from scalability concerns. The schedule search space for anything but toy computational kernels are large enough to make it impractical to explore it iteratively and try out different schedules by running them. Even in cases where the search space is not so large as to make the search completely impractical, it is still too time consuming to be palatable to developers in practice. The analytic models, on the other hand, either fail to account for features specific to hardware architecture, thus failing to extract maximum performance, or are tied to particular architectures and can become obsolete as the architecture evolves.

3.2 Using Machine Learning to Select Optimizations

Machine learning, using both static and dynamic features, has been used to select optimization parameters in various contexts within compilers and code optimization: parameters for specific loop optimizations, compiler command line invocation parameters, and compiler optimization pass selection.

Stephenson et al. [29] use machine learning to select a loop unroll factor. They explore two machine learning approaches: Near Neighbour (using the Euclidean distance between feature vectors as the similarity vector) and SVM. They find that their best classifier comes within 7% of the optimal unroll factor 79% of the time, obtaining a 5% speedup over the Open Research Compiler baseline with software pipelining disabled and a 1% speedup

with software pipelining enabled. Stephenson et al. formulate a multi-class classification problem: given a set of program features, which one of eight loop unroll factors (1, 2,. . . ,8) yields the best performance? The features used included features such as the number of operands in the loop body, the number of branches in the loop body, the number of indirect references in the loop body, and the cycle length of the loop body. Of the two machine learning approaches, SVM performs better than Near Neighbour. Stephenson et al. use feature selection to narrow down their initial list of 38 features. They explore two feature selection techniques: Mutual Information Score (MIS) and greedy feature selection. MIS attempts to quantify the reduction in uncertainty regarding the value of the loop unroll factors given the value of a particular attribute. The features with the highest MIS values are chosen. Greedy feature search chooses a feature at a time in the order in which the features reduce the classifier error. Unlike MIS, greedy feature search takes into account the effect of multiple features in tandem. The union of the features selected by the two algorithms were used to generate the final classifier.

Cavazos et al. [6] utilize machine learning to select compiler optimization settings, using standard x86/x86_64 architecture hardware performance counters as input features. The performance counters reflect the number of memory accesses, cache hits and misses at all cache levels, translation lookaside buffer statistics, floating point instruction statistics, branch instruction statistics, and CPU cycle statistics. Logistic regression, a probabilistic approach, is used to find a set of compiler optimizations to allow for optimal performance. Cavazos et al. [6] found that their model results in a set of transformations which allow the compiler to compile binaries that run substantially faster than when the default aggressive optimization settings are used.

Fursin et al. [14] describe MILEPOST GCC[1], an adaptation of the GCC compiler. MILEPOST GCC utilizes machine learning to determine which optimization passes to use when compiling code. The features used are static features extracted from the source code of the program being optimized (e.g., number of conditional branches, number of assignment instructions, number of unary operations, number of calls with pointers as arguments). Programs for the MiBench benchmark suite are used for testing. A training set of 500 sequences of compiler flags and their values, which were mapped to optimization passes, was generated by uniform random selection from the overall space of flag settings. Each of the compiler flag sequences was used to compile each of the benchmarks used for training, and a speed-up for each benchmark relative to a baseline was recorded. From these results, a mapping from the training benchmark features to a *distribution over good solutions* was learned. For a test (unseen) program, its features were extracted, and the learned good solution distribution was used to predict the best set of optimization passed to be used. The prediction process is one-shot, without the need for an iterative search through the solution space. Fursin et al. found that their adaptive optimization approach provides, on average, an 11% runtime improvement over the optimization passes GCC uses by default with the -O3 optimization setting.

Yuki et al. [34] apply machine learning to the loop optimization search space with a focus on loop tiling. The primary contribution is the set of features selected to select

[1] `http://ctuning.org/wiki/index.php/CTools:CTuningCC`

an optimal tile size. Specifically, for every source program, the number of read and write operations in the innermost loops is added up and classified according whether the memory operation could make efficient use of the CPU's linear prefetch capability. Accordingly, memory operations were classified as either *RP* (prefetched reads), *NP* (non-prefetched reads), *RI* (invariant reads), *WP* (prefetched writes), *WNP* (non-prefetched writes), and *WI* (invariant writes). A memory operation was considered *invariant* if the target memory location was invariant with respect to the loop iteration; it was considered *prefetched* if the CPU could successfully prefetch into cache memory regions addressed in later loop iterations based on accesses in prior loop iterations, e.g., accessing consecutive words in memory; and it was considered *non-prefetched* if linear prefetching could not fetch into cache memory regions accessed in later iterations, e.g., random access. Yuki et al. find that the tile sizes selected by using the Artificial Neural Networks machine learning algorithm when using their features of choice fall within 5% of optimal tile sizes. The features used by Yuki et al. are also included in the features used to construct the classifier in my work.

The previous work discussed in this section focuses either on coarse-grained classes (selecting compiler command line options or compiler optimization passes) or on optimizing a single loop transformation in isolation (loop unrolling or loop tiling). The generation of a search space consisting of compositions of transformations and selecting the best transformation composition are not explored.

3.3 Machine Learning and the Polyhedral Model

Park et al. [23] use machine learning to predict the best polyhedral transformation for a candidate source program. The search space of candidate polyhedral transformations is limited to transformations of several types supported out-of-the-box by the the Polyhedral Compiler Collection (PoCC) [2] software: loop tiling, loop fusion, loop parallelization, loop pre-vectorization, and loop unrolling. The transformation types are, in theory, independent of each other and can be applied in all combinations. The cross product of them all would result in a search space of several hundred transformations. In reality, my experience suggests that the transformations are not always independent, and not all are well-supported. This will be discussed in greater detail in Chapter 4. Like Cavazos et al. [6], Park et al. use hardware performance counters as features. However, the precise list of features used is not specified.

Park et al. endeavour to predict the runtime of a source program that has been subjected to a sequence of defined polyhedral transformations. Two approaches are then evaluated to select the optimal transformation to use. In the *one-shot* approach, the transformation corresponding to the lowest runtime is selected. In the *five-shot* approach, the transformations corresponding to the five best runtimes are applied to the source and each resulting program is compiled and run. The program with the best actual runtime of the five is selected. Two machine learning techniques to predict runtimes are evaluated: linear regression and Support Vector Machine (SVM) regression.

[2] `http://www.cs.ucla.edu/~pouchet/software/pocc`

Park et al. [23] find that that using the Polyhedral model for loop optimization produces runtimes 2–3.5× better than those achieved by merely optimizing compiler flags. Moreover, they report their method outperforms the static method utilized by Bondhugula et al. [4]. Evaluating linear regression and SVM, they find that the former yields a runtime improvement of 3.16× in the one-shot trial and 3.50× in the five-shot trial, while the runtime improvement produced by the latter is 3.27× in the one-shot trial and 4.68× in the five-shot trial. If the optimal transformation is defined as the transformation in the search space that produces the best runtime, the transformation chosen in the one-shot trial yields a runtime within 63% of the optimal on average, while the transformation chosen in the five-shot trial yields a runtime within 84% of the optimal on average. The findings by Park et al. are discussed in greater detail in Chapter 5.

To achieve good performance with the technique proposed by Park et al., it is necessary to run the program being optimized up to five times. For anything but trivial programs with small input sizes, this may prove to be excessively time consuming and would serve as a barrier to the uptake of the technique. This thesis builds on the work by Park et al., expanding the set of features to include static features in addition to dynamic features, formulating the problem as a classification rather than a regression problem to make it easier to learn, performing feature selection, and using classifier ensembles to improve classification accuracy.

3.4 Summary

In this chapter, I discussed prior work in the fields of loop optimizations in the polyhedral model and applying machine learning to loop optimizations. Since a method to select one-dimensional and multi-dimensional schedules for simple loop nests was introduced over two decades ago, research has been done into extending schedule generation to larger classes of programs and generating schedules and loop transformations within a reasonable timeframe. Machine learning has been used to select compiler arguments to produce faster code, as well as select loop transformations to apply to source code before passing it through the compiler.

Chapter 4

My Proposal

In this chapter, I discuss my proposal for learning rules to select efficient loop nest transformations for computational kernels. I first discuss the initial set of features constructed potentially to be predictive of the effect of different transformation combinations on different computational kernels. I then proceed to discuss the procedure used to collect data and the approach used for evaluation. Finally, I conclude by discussing how the feature selection was used to reduce the initial feature set to a subset found to be relevant to the problem at hand.

My proposal is an extension of the work by Park et al. [23]. I use classification machine learning methods rather than regression models, and to that end formulate the problem as a classification problem rather than a regression problem. I augment the set of features with static features in addition to the dynamic features derived from hardware performance counter values. I perform feature selection to exclude features that hold little or no predictive value for the class. Finally, I use classifier ensembles to improve classification accuracy.

4.1 Initial Feature Set

I start out considering two sets of features. The first set consists of features that are hardware performance counter values, collected during a run of a binary compiled from the unmodified program source. Hardware performance counter values are commonly available in modern microprocessors and can be used in performance tuning and analysis. The second set consists of features that can be extracted from analyzing the program source structure, in particular features that characterize the program's memory access patterns.

4.1.1 Hardware Performance Counters

Modern microprocessors are complex and include advanced pipelining, prefetching, branch prediction, and instruction level parallelism functionality. This functionality continuously

evolves as CPU manufacturers unveil new architectures. Furthermore, the full details of the CPU architecture is typically a trade secret and therefore not well-documented. As a result, several well-known rules of thumb aside, it can be difficult to ascertain how a fragment of source code, or even assembly, will run on a CPU and how changes to the code structure will affect runtime. In order to make it easier to diagnose performance issues, modern CPUs have hardware performance counter functionality that can be used to shed light on the inner workings of the CPU. The counters track discrete events in the microprocessor pertaining to the operation of the hardware, such as instruction flow, memory access, and branches. Sampling the counters before and after a program executes allows us to partially characterize the interactions of the program with the CPU.

Park et al. [23] also use performance counters as machine learning features in their work on selecting a good polyhedral transformation for a computational kernel. However, they do not explicitly name the counters that were used. My starting point is counters related to clock cycles, memory cache events, stalls while waiting for resources, and branch prediction events. Clock cycle counters allow us to capture the overall CPU time cost of the program. Memory cache misses are costly, as cache layers closer to the CPU are faster to access than cache layers that are farther way. A cache miss in a faster cache level requires a slower memory access in the next level. Resource stalls have a detrimental effect on runtime since the CPU is unable to do useful computational work while waiting for a resource request to be satisfied. Branch prediction attempts to correctly predict which branch of a conditional instruction will be taken. Instructions from the predicted branch are then loaded into the CPU's pipeline. If the prediction is wrong, the pipeline must be flushed, and instructions from the other branch must be loaded instead. As CPU pipelines grow deeper and more complex, mispredicted branches exact an increasingly higher cost in terms of wasted cycles. The raw performance counters captured are listed in Table 4.1.

Raw counter values cannot be used as features, since higher counter values can be indicative of repetitive execution rather than reflect the program structure. Consider executing an identical loop for 100 iterations and for 200 iterations. The instance that executes for 200 iterations is likely to have higher counter values across the board, even though both programs are structurally identical. Therefore, the raw counters are scaled to generate counter features. Cache misses for a cache layer are divided by the total number of cache accesses for that layer. Resource stall values are scaled with respect to the overall number of clock cycles. Values pertaining to instruction counts are scaled with respect to the total number of instructions retired (instructions executed to completion), resulting in feature values in the range $[0, 1]$. The normalized feature values are then discretized. The discretization was carried out manually for the purposes of this work, but it could be easily automated. The counter features selected are listed and described in Table 4.2.

4.1.2 Memory Operation Features

Further features were added to capture the type of memory accesses in the innermost loops, as proposed by Yuki et al. [34] in their work on machine learning for loop unrolling. Innermost loop memory accesses are classified along two dimensions: access type (read or

Table 4.1: Performance Counter Measurements Collected

counter	description
CPU_CLK_UNHALTED:TOTAL_CYCLES	Total number of core cycles
PAPI_L1_TCA	L1 cache accesses
PAPI_L1_TCM	L1 cache misses
PAPI_L2_TCA	L2 cache accesses
PAPI_L2_TCM	L2 cache misses
PAPI_L3_TCA	L3 cache accesses
PAPI_L3_TCM	L3 cache misses
MEM_LOAD_RETIRED:LLC_MISS	Last level cache misses
RESOURCE_STALLS:ANY	Resource related stall cycles
RESOURCE_STALLS:LOAD	Load buffer stall cycles
RESOURCE_STALLS:RS_FULL	Reservation Station full stall cycles
RESOURCE_STALLS:ROB_FULL	Re-order Buffer full stall cycles
INSTRUCTION_RETIRED	Number of instructions at retirement
BR_INST_EXEC:COND	Conditional branch instructions executed
BR_MISP_EXEC:ANY	Mispredicted branches executed
UOPS_RETIRED:STALL_CYCLES	Cycles no micro-ops retired

write) and interaction with the automatic prefetcher. Automatic prefetchers in modern CPU detect a sequence of cache misses with a consistent stride, with either increasing or decreasing memory addresses, and automatically prefetch memory at the detected stride interval, often up to a page boundary. This automatic optimization speeds up operations like accessing array elements in a tight loop. Accordingly, memory accesses are characterized as prefetchable (occurring with a fixed stride), non-prefetchable (lacking a fixed stride pattern), or loop-invariant (the same address being accessed in all loop iterations). Combining the two dimensions gives us the following features: the number of instances of prefetched memory reads, the number of instances of non-prefetched memory reads, the number of instances of loop-invariant memory reads, the number of instances of prefetched memory writes, the number of instances of non-prefetched memory writes, and the number of instances of loop-invariant memory writes. In my experiments, the feature values for each benchmark were manually determined. However, this feature extraction process could be automated using, for example, techniques described by Fursin et al. [14]. The read values are normalized with respect to the overall number of memory reads, as determined from the performance counter values, yielding a value in the range [0, 1]. The write values are similarly normalized with respect to the number of writes. The normalized values are discretized manually. As mentioned previously, the discretization process can be automated in future work. The list of features is also captured in Table 4.3.

The vector $\langle c_0, \ldots, c_{12}, m_0, \ldots, m_5 \rangle$ is then referred to as the per-benchmark *feature vector*.

Table 4.2: Performance Counter Features

feature	counter expression
c_0	the ratio of L1 cache misses to the number of L1 cache accesses
c_1	the ratio of L2 cache misses to the number of L2 cache accesses
c_2	the ratio of L3 cache misses to the number of L3 cache accesses
c_3	the ratio of total CPU cycles to the number of retired instructions
c_4	the ratio of cycles when no instructions were retired to the number of total CPU cycles
c_5	the ratio of L3 cache accesses to the number of total CPU cycles
c_6	the ratio of L3 cache misses to the number of total CPU cycles
c_7	the ratio of all resource-related stall cycles to the number of total CPU cycles
c_8	the ratio of load buffer stall cycles to the number of total CPU cycles
c_9	the ratio of stall cycles due to the reservation station being full to the number of total CPU cycles
c_{10}	the ratio of stall cycles to the reorder buffer being full
c_{11}	the ratio of conditional branch instructions executed to the total number of retired instructions
c_{12}	the ratio of mispredicted branches executed to the total number of retired instructions

Table 4.3: Memory Access Features

feature	description
m_0	number of prefetched memory reads
m_1	number of non-prefetched memory reads
m_2	number of loop-invariant memory reads
m_3	number of prefetched memory writes
m_4	number of non-prefetched memory writes
m_5	number of loop-invariant memory writes

4.2 Class Value

Supervised machine learning algorithms generally fall into one of two categories: regression and classification. Regression algorithms predict continuous, real-valued outcomes. Classification algorithms, on the other hand, predict discrete-valued classes. Binary classification is a special case of classification, where the class has only two values. The properties of the class to be learned guide the choice of the machine learning algorithm.

The work by Park et al. [23] uses linear regression and a regression variant of SVM to predict the ratios of the runtimes of programs subjected to a sequence of loop transformations to the runtime of the unmodified program (in other words, the speedup or slowdown caused by applying the transformation). The program predicted to have the fastest runtime indicates the best sequence of transformations to apply. Regression is used since a continuous value is being predicted directly.

The approach in this thesis is to instead express the problem as a classification problem. Two benefits to the classification approach are hypothesized: that it will make the problem easier to learn and reduce the error rate. While it can be argued that a linear regression evaluation can also be easily incorporated in tools like compilers, both the one-shot and the five-shot linear regression variants yield relatively poor results, and SVM regression was required for satisfactory speed improvements. Unlike linear regression, SVM regression evaluation is complex. To formulate the classification problem, I consider which one of any two sequences of transformations is preferable for a given source program. One sequence of transformations is preferable to another if a program that has had that sequence of transformations applied to it has a faster runtime than a program that has had the other sequence of transformations applied. I then attempt to use machine learning techniques to predict which one of any two sequences of transformations is preferable for an arbitrary program.

More formally, given a source program P and two transformation sequences, T_A and T_B, the transformation sequence pair (T_A, T_B) is labelled 1 if P has a faster runtime having had T_A applied rather than having had T_B applied. Otherwise, if P has a faster runtime having had T_B applied rather than having had T_A applied, the transformation sequence pair is labelled 0. A row in the training data consists of the feature values for the benchmark, the transformation sequence pair, and the class of the transformation sequence pair.

Once the classifier is constructed, it is used to choose a transformation sequence as follows. For a previously unseen program P, our machine learning algorithm attempts to label all transformation sequence pairs (T_A, T_B) for all transformations A and B in the transformation search space. Subsequently, for every transformation sequence τ, the number of times a tuple (τ, T), where T is any transformation other than τ, is predicted to have the label 1 is recorded as $score_\tau$. The transformation sequence with the highest score is selected as the best transformation suggested by the machine learning algorithm. In effect, a voting algorithm is used, with transformations being voted on within the context of every transformation sequence pair combination. The transformation sequence with the most votes is considered the winner. This is referred to as *pairwise preference ranking* or *round robin ranking* [13]. Other approaches to combining pairwise preferences exist (e.g., algorithms to calculate class probabilities [33]) but are not explored in the context of this thesis.

A binary classification problem easily lends to learning using decision trees. The simplicity of decision trees makes them amenable to be easily translated to code and subsequently incorporated into optimizing compilers.

4.3 Data Collection

The benchmarks used to train and test the classifier are taken from Polybench/C [1], a benchmark suite that consists of computational kernels with loop nests that meet the static

[1] `http://www.cs.ucla.edu/~pouchet/software/polybench`

control parts limitations imposed by the polyhedral model. The PAPI library [2] is used to collect performance counter values for each benchmark. Each benchmark is compiled using the `gcc` compiler version 4.7.2 [3] at the highest standard optimization level, $-$O3 for `gcc`. No other `gcc` flags are used. The benchmarks are then executed. The wall time runtime and selected hardware performance counter values are recorded. Each benchmark is run enough times for the sum of the runtimes of the executions to reach at least 10 seconds. The number of times each benchmark is run is also recorded.

Subsequently, the Polyhedral Compiler Collection (PoCC) [4] is used to generate the transformation sequence search space. PoCC has numerous options for various optimizations. Similarly to Park et al. [23], I consider the following transformation types:

- *Fusion.* PoCC supports three fusion settings: nofuse, smartfuse, and maxfuse. However, experimentation showed that these settings yield unpredictable and inconsistent results when applied to different benchmarks. For example, it might be expected that when nofuse is chosen, no fusion transformations at all are applied. This is, indeed, the case with some benchmarks. With other benchmarks, on the other hand, nofuse produces the same fusion transformations as smartfuse, while with others still, nofuse results in some loops being fissioned. The maxfuse setting sometimes has similarly unpredictable results. The lack of consistency resulted in noisy data that makes it difficult to learn a classifer. To deal with the inconsistencies, the fusion setting is always fixed to smartfuse, except in the case of the identity transformation.

- *OpenMP.* The OpenMP setting is a binary on or off choice. Either OpenMP is enabled, resulting in the generation of parallel code, or it is disabled, resulting in no parallelism.

- *Tiling.* Tiling can be applied at any or all of the first three loop nesting levels. The tile size at each level could be 1 or 32. This results in $2^3 = 8$ tiling settings.

- *Vectorization.* The vectorization setting is similarly binary, on or off. As a caveat, however, PoCC only supports enabling vectorization if tiling is enabled for at least one loop level.

- *Loop unrolling.* Three choices for loop unrolling were available: no loop unrolling, a loop unrolling factor of 4, and a loop unrolling factor of 8.

In addition, the identity transformation is also included as a possible transformation. The identity transformation does not alter the original source code in any way.

Adding up the identity transformation to the product of two choices for OpenMP, eight choices for tiling, two choices for vectorization, and three choices for loop unrolling, and then subtracting the invalid combination of vectorization and no tiling, we have 96 transformations in total.

[2] `http://icl.cs.utk.edu/papi/index.html`

[3] `http://gcc.gnu.org/gcc-4.7`

[4] `http://www.cs.ucla.edu/~pouchet/software/pocc`

PoCC generates one source file per transformation sequence per benchmark. The source files are then all compiled, generating a binary per transformation sequence per benchmark. Each binary is executed, and the wall time runtime is recorded. A row of data is then generated for every pair of transformation sequences in each benchmark. The format of the row is c_0, c_1, ..., c_{12}, m_0, m_1, ..., m_6, $identity_0$, $tiling_0$, $openmp_0$, $vectorization_0$, $unroll_0$, $identity_1$, $tiling_1$, $openmp_1$, $vectorization_1$, $unroll_1$, $class$. The features c_0, c_1, ..., c_{12} are the discretized performance counter values for the benchmark, and m_0, c_1, ..., m_6 are the memory feature values for the benchmark. Attributes with subscript 0 pertain to the first transformation sequence in the pair, while attributes with subscript 1 refer to the second. *Identity* is a binary attribute, and its value is 1 if the transformation is an identity transformation and 0 otherwise. *Tiling* is an enumerated attribute corresponding to the tiling setting used in the transformation. *Openmp* and *vectorization* are binary attributes reflecting the presence or absence of those optimization in the transformation. The *unroll* value of unroll is the loop unrolling factor used (8, 4, or 0 for no unrolling). The feature *class* is the binary classification. If the runtime of the first transformation in the pair is less than the runtime of the second transformation, *class* is 0. Otherwise, it is 1. The transformations t_0, t_1 and t_1, t_0 are treated as distinct pairs, and a separate data row is generated for each. This is done so as to create a balance in the class values in the training data and to avoid overfitting. Of course, the two pairs will belong to opposite classes. The benchmark features are combined with the per-transformation rows to generate the complete benchmark dataset. Anywhere from one to all of the features may be used.

Once the data is collected, a leave-one-out approach is used to generate decision trees and to test their effectiveness. For each benchmark, the data sets of all other benchmarks are used as the training data, while the data set of the benchmark in question is used as the test data. C4.5 [5] is used to generate a decision tree from the training data. That decision tree is then used to classify the test data. The classifications produced by the decision tree are then processed, with each transformation scored according to how it compares with other transformations in the test data. For every row, t_0 is awarded one point if the class assigned to the row is 0 (the runtime of t_0 is predicted to be lower than the runtime of t_1), and t_1 is awarded one point if the class of the row is 1. The transformation are then ordered in descending order according to the number of points awarded to them, and this ordering reflects the effectiveness of the transformations for the benchmark as predicted by the decision tree, from best to worst. Ties are broken randomly. To determine the value of the prediction of the decision tree, the runtime of the top-ranked transformation is looked up, and the ratio of that runtime to the best runtime of all the transformations is reported. This is referred to as the one-shot result. Subsequently, the runtime values of the top five transformations are looked up, and the ratio of the best of the five runtimes to the overall best runtime is reported as the five-shot result.

[5] `http://www.rulequest.com/Personal`

4.4 Feature Selection

A classifier constructed using all features in the initial set can prove to have an unsatisfactorily high error rate, possibly due to the presence of irrelevant or redundant features, which makes constructing an effective classifier more difficult. Two approaches to feature selection [16] are explored.

In the first approach, a classifier is generated using every individual feature and every combination of two features. The classifier is evaluated for every benchmark. The runtime of the transformation sequence obtained as a result of the scoring method described in the previous section is compared to the expected runtime of a randomly-selected transformation sequence, obtained using the formula,

$$\frac{1}{T}\sum_{t=1}^{T} runtime_t,$$

where T is the total number of transformation sequences for the benchmark and $runtime_t$ is the runtime of the binary generated by applying transformation t. If applying the transformation sequence suggested by the classifier results in a slower runtime than the expected runtime of a transformation selected at random, the classifier is considered to have failed. A feature is discarded if none of the classifiers generated from it is deemed successful. This feature selection approach proved ineffective as it turned out that each feature combination was considered successful for at least one benchmark, which meant that the approach failed to exclude any features.

In the second approach, a classifier is again generated using every individual feature and every combination of two features and evaluated for every benchmark. The evaluation is performed similarly to before, by comparing the runtime of a program generated by applying the transformation sequence suggested by the classifier to the expected runtime of a program generated by applying a randomly selected transformation sequence, calculated as in the first approach. The classifiers are then sorted in order of the number of benchmarks where the classifier suggested a transformation sequence with a better runtime than the expected runtime of a randomly selected transformation sequence for the benchmark. For example, a classifier considered to be successful for 16 benchmarks and unsuccessful for 14 benchmarks will be ranked ahead of a classifier considered successful for 14 benchmarks and unsuccessful for 16 benchmarks. As shown in Chapter 5, this approach to feature selection proves effective. The features that were included in the combinations selected by this approach are listed in Table 4.4.

The two features that appear the most often are the number of L1 cache misses normalized with respect to the number of L1 cache accesses and the number of L3 cache accesses normalized with respect to total CPU cycles. On the other hand, the number of L3 cache accesses normalized with respect to L3 cache misses does not appear at all. It is not immediately obvious why L1 cache misses are significant while L3 cache misses are not and why overall L3 cache accesses are significant while overall L1 cache accesses are not. This may be related to architectural peculiarities. It is also interesting to note that

Table 4.4: Selected Features

feature	description	number of appearances
c_0	ratio of L1 cache misses to L1 cache accesses	8
c_1	ratio of L2 cache misses to L2 cache accesses	4
c_3	ratio of total CPU cycles to the number of retired instructions	3
c_4	ratio of cycles when no instructions were retired to total CPU cycles	1
c_5	ratio of L3 cache accesses to total CPU cycles	6
c_6	ratio of L3 cache misses to total CPU cycles	2
c_7	ratio of all resource-related stall cycles to total CPU cycles	1
c_{12}	ratio of mispredicted branches executed to total retired instructions	3
m_1	number of non-prefetched memory reads	4
m_3	number of prefetched memory writes	3
m_4	number of non-prefetched memory writes	2
m_5	number of loop-invariant memory writes	1

prefetched memory writes are significant, while prefetched memory reads are not. Again, this may be related to the architecture on which the experiment was run. Such subtleties regarding feature significance are not intuitive, and they become apparent only as a result of the feature selection process.

4.5 Classifier Ensembles

Two classifier ensemble construction approaches are explored.

The first approach uses simple voting, with the top n classifiers as determined in the feature selection stage voting on the classes of the test data. Each classifier's vote is weighted equally. As shown in Chapter 5, this ensemble construction performs very well. A value of 20 is used for n. The value is determined empirically, and it is found that a value of 20 performs better than a lower value, while values between 20 and 40 all perform relatively equally well.

Another alternative considered is the final classifier being the result of a vote of the top n classifiers. The classes selected by the top n classifiers become features used to construct a new decision tree classifier with C4.5. The format of the data rows for the training and test data used to construct and evaluate this new classifier is c_1, c_2, ..., c_n, $identity_0$, $tiling_0$, $openmp_0$, $vectorization_0$, $unroll_0$, $identity_1$, $tiling_1$, $openmp_1$, $vectorization_1$, $unroll_1$, $class$. c_1, c_2, ..., c_n are the class values predicted for the transformation sequences by the top n classifiers from the previous stage. The other attribute values represent the transformations applied and the actual class, as before. This approach was found to produce inferior results to simple voting and was therefore discarded.

4.6 Complete Classifier Construction

The end-to-end proposed classifier construction process is described in Algorithm 4.1.

Algorithm 4.1: Classifier Construction

foreach *combination of one and two features from* c_0, c_2, ..., c_n **do**
  foreach *benchmark* **do**
    Construct a training set from data for all benchmarks excluding *benchmark* using the feature combination;
    Construct a test set from data for benchmark *benchmark* using the feature combination;
    Learn a classifier using the constructed training set;
    Determine the effectiveness of the classifier for the test set, as described in the section on feature selection;
  end foreach
end foreach
Tally up the number of benchmarks for which the classifier using the feature combination was judged effective, as described in the section on feature selection;
Sort feature combinations in descending order according to the number of benchmarks for which they were judged effective;
Take top 20 feature combinations and construct a decision tree from each combination;
Classify the data using the 20 trees; determine the final class by a vote of all 20 trees;

4.7 Summary

In this chapter, I proposed an approach to learning the best loop transformation sequence among a finite transformation sequence search space for an arbitrary computational kernel. I listed the features used as an input to the machine learning algorithm, formulated the task as a binary classification problem, and described the process of data collection, decision tree construction, and feature selection.

Chapter 5

Evaluation of My Proposal

In this chapter, I present an evaluation of my proposal against a baseline and the current state-of-the-art approach. I describe the baseline used first. I then proceed to describe my reproduction of the approach and the results from the work by Park et al. [23]. Finally, I describe the results obtained using my proposed approach: a vote of multiple decision tree classifiers.

In all cases, the results are reported in two ways. First, I present the percentage-of-optimal results, the ratio of the runtime of the benchmark binary obtained by applying the optimal transformation sequence in the search space to the runtime of a benchmark binary obtained by applying a transformation sequence selected by the approach being evaluated. The ratio is expressed as a percentage, with a value of 100% indicating that the transformation sequence selected is, in fact, the optimal transformation sequence in the search space. The optimal transformation in the search space is determined by applying every transformation sequence in the search space to the benchmark source code, running the resulting binaries, recording the runtimes, and selecting the transformation sequence corresponding to the binary with the lowest runtime. Second, I present the results as the ratio of the runtime of the untransformed, or stock, program to the runtime of a benchmark binary obtained by applying a transformation sequence selected by the approach being evaluated. This ratio represents the speed-up factor of the transformation selected by the classifier over the untransformed program. The transformed program is faster than the stock program if the speed-up factor is greater than 1; conversely, the stock program is faster than the transformed program if the speed-up factor is less than 1.

5.1 Baselines

Two baselines are used:

1. The runtime of the stock benchmark with no modifications to the source code. In the subsequent tables and graphs, this value is referred to as *identity*.

2. The expected runtime of a randomly-selected transformation sequence, obtained using the formula,

$$\frac{1}{T}\sum_{t=1}^{T} runtime_t,$$

where T is the total number of transformation sequences for the benchmark and $runtime_t$ is the runtime of the binary generated by applying transformation t. In the subsequent tables and graphs, this value is referred to as `expected-random`.

5.2 Reproduction of Prior Work

The reproduction of the results by Park et al. [23] of the approaches utilizing linear regression and SVM tries to hew as closely as possible to the methodology outlined in their paper. The performance counters used as features are listed in Table 5.1. All counter values are normalized by dividing them by the number of instructions in each benchmark, obtained by querying the counter PAPI_TOT_INS. The transformation sequence search space used is the reduced search space as described in Chapter 4, needed to accommodate interactions between different transformations and apparent bugs in the PoCC software.

The performance counter values and runtimes are collected as described in Chapter 4. A leave-one-out approach is used to evaluate the efficacy of the machine learning approaches, with the data for one benchmark used for evaluation and the data for the other benchmarks used for training. The leave-one-out process is repeated once for every benchmark, so that error rates for all benchmarks can be calculated. For every benchmark, the training set consists of an entry for every transformation sequence in the search space and has the format c_0, c_1, ..., c_{12}, *tiling*, *openmp*, *vectorization*, *unroll*, *class*. c_0, c_1, ..., c_{12} are the continuous, non-discretized, normalized performance counter feature values for the benchmark. *Tiling* is an enumerated attribute corresponding to the tiling setting used in the transformation. *Openmp* and *vectorization* are binary attributes reflecting the presence or absence of those optimization in the transformation. The *unroll* value of unroll is the loop unrolling factor used (8, 4, or 0 for no unrolling). *class* the ratio of the runtime of the benchmark binary obtained by applying the optimal transformation sequence in the search space to the runtime of a benchmark binary obtained by applying the transformation sequence corresponding to the row. In other words, the class represents the speed-up or slowdown of the transformed benchmark relative to the original benchmark. The class is continuous. Weka [1] 3.7.5 is used for both training and evaluation. Default parameters are used both for linear regression and for SVM. The output of Weka is predicted runtime ratios for every transformation sequence for the benchmark used for testing. The predicted ratios are then sorted in ascending order, from smallest to largest, and this relative order is taken to be the order of the transformation sequences from best to worst as predicted by the machine learning process. The class values predicted by Weka are ignored except to determine this order. In fact, in some cases the class values predicted by Weka were negative.

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

5.3 Evaluation Methodology for My Approach

A leave-one-out approach [17, pp. 245-247] was used to evaluate the classifier, feature selection approach, and approach to constructing the classifier ensemble. The overall evaluation algorithm for the second approach is described in Algorithm 5.1.

Algorithm 5.1: Feature Selection and Classifier Evaluation

```
foreach benchmark do
    foreach combination of one and two features from c_0, c_2, ..., c_n do
        foreach benchmark' excluding benchmark do
            Construct a training set from data for all benchmarks excluding
            benchmark' and benchmark using the feature combination;
            Construct a test set from data for benchmark benchmark' using the
            feature combination;
            Learn a classifier using the constructed training set;
            Determine the effectiveness of the classifier for the test set;
        end foreach
        Tally up the number of benchmarks for which the classifier using the feature
        combination was judged effective;
    end foreach
    Sort feature combinations in descending order according to the number of
    benchmarks for which they were judged effective;
    Take top 20 feature combinations and construct a decision tree from each
    combination;
    Construct a final classifier that classifies the data using the 20 trees and
    determines the final class by a vote of all 20 trees;
    Use final classifier obtained after feature selection to predict best transformation
    sequence for benchmark;
    Determine the effectiveness of the final classifier for benchmark and report this
    value;
end foreach
```

5.4 Discussion

For both linear regression and SVM, one-shot and five-shot results are reported. One-shot results evaluate the transformation sequence predicted to be the best by the machine learning process. Five-shot results take the best runtime of the top five results as predicted by the machine learning process. The binary corresponding to each of the top five results is run and its runtime is recorded. The binary that has the best runtime is used.

Table 5.2 contains percentage-of-optimal results for all benchmarks for the identity transformation; the expected random selection; and the one-shot and five-shot approaches for each of linear regression, SVM regression, and decision tree voting. A graph showing a comparison of the baseline, one-shot, and five-shot results for both linear regression and

SVM is shown in Figure 5.1. The results of the reproduction are similar to the results reported by Park et al. SVM produces better results than linear regression, while the five-shot approach outperforms the one-shot approach. With the transformations in the search space ranked in reverse runtime order, the transformation selected using one-shot linear regression is, on average, in the 73rd percentile; the transformation selected using five-shot linear regression is, on average, in the 79th percentile; the transformation selected using one-shot SVM regression is, on average, in the 70th percentile; the transformation selected using five-shot SVM is, on average, in the 87th percentile; and the transformation selected using one-shot decision tree voting is, on average, in the 86th percentile. As previously mentioned, the five-shot approach necessitates running the candidate program five times, a process which, depending on the runtime of the program, can be unduly time consuming. The classification and prediction process for each benchmark takes between 0.5 seconds to 0.7 seconds. This includes the classification time for the 20 trees, the time it took to convert the pairwise preferences to rankings, and the time it took tally the votes of the 20 classifiers and come up with the final prediction. The derivation of the rankings and the voting are implemented as proof-of-concept scripts, and an efficient implementation would likely make the process take even less time. On the other hand, the runtime of each benchmark when run with an input size of 1024 elements is between 0.5 seconds and 20 seconds, depending on the benchmark. For most benchmarks, the decision tree vote prediction process takes less time than a single benchmark run, resulting in substantial time savings over a 5-shot approach.

Figure 5.2 contrasts the percentage-of-optimal one-shot decision tree vote results with the identity and the expected random selection baselines, as well as with the five-shot linear regression and SVM results. One-shot decision tree voting performs significantly better than the identity transformation, the expected runtime for a randomly selected transformation, and the five-shot linear regression approach. It performs about as well as the five-shot SVM approach, with the benefits to the decision tree approach being the lack of necessity to run the candidate program multiple times and the ability to more easily represent the classifier as a sequence of conditional statements.

The per-benchmark breakdown in Table 5.2 reveals that while, on average, the one-shot decision tree outcome is about as good as the five-shot SVM outcome (and therefore as good as or better than prior state-of-the-art approaches), the different approaches do better on different benchmarks. For instance, the decision tree approach gets the optimal transformation sequence (100%) for the `covariance` benchmark, while the second best result for that benchmark is 77% with linear regression. Similarly, decision tree voting gets 100% for `seidel-2d`, while the second closest is SVM with 61.4% (both one-shot and five-shot). On the other hand, decision tree voting gets only 55.0% on `fdtd-2d`, while both one-shot and five-shot linear regression get 98.7%. Similarly, one-shot decision tree vote gets 68.2% for `jacobi-2d-imper`, while one-shot linear regression gets 80.3% and one-shot SVM gets 98.8%. Examining benchmarks where each approach excels, one-shot linear regression scores over 90% nine times, five-shot linear regression 13 times, one-shot SVM 12 times, five-shot SVM 19 times, and one-shot decision tree voting 18 times.

Table 5.3 contains speed-up results over the identity transformation for every benchmark for expected random selection as well as the one-shot and five-shot approaches for

each of linear regression, SVM regression, and decision tree voting. Figure 5.3 presents these results in bar chart form. The three benchmarks with the largest speed-ups appear in a separate chart with a different scale so as not to make the results for benchmarks with more modest speed-ups unreadable. The one-shot decision tree vote speed-up is, on average, slightly better than the five-shot SVM regression speed-up. However, this result is not uniform across all benchmarks. Decision tree voting comes very close to the SVM speed-up result for most benchmarks, outperforming it on some benchmarks and underperforming on others. These results show that `gcc`, even at the highest optimization setting, does not currently do a satisfactory job at loop optimization and that there are substantial speed-ups to be gained by employing a polyhedral model optimization pass.

Table 5.1: Performance Counter Measurements Collected for Linear Regression and SVM Predictions

counter	description
PAPI_L1_TCA	L1 cache accesses
PAPI_L1_TCM	L1 cache misses
PAPI_L2_TCA	L2 cache accesses
PAPI_L2_TCM	L2 cache misses
PAPI_L3_TCA	L3 cache accesses
PAPI_L3_TCM	L3 cache misses
PAPI_TLB_TL	Total translation lookaside buffer misses
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_RES_STL	Cycles stalled on any resource
PAPI_TOT_IIS	Instructions issued
PAPI_VEC_SP	Single precision vector/SIMD instructions
PAPI_VEC_DP	Double precision vector/SIMD instructions

Table 5.2: Linear Regression, SVM, and Decision Tree Vote: Percentage of Optimal

Benchmark	Identity	Expected Random	Linear Regression		Support Vector Machine		Decision Tree Vote
			1-shot	5-shot	1-shot	5-shot	
2mm	15.9	44.6	76.9	100.0	96.7	100.0	77.8
3mm	19.0	45.9	99.4	99.4	75.1	89.0	100.0
adi	73.3	35.8	46.5	54.1	40.8	48.8	100.0
bicg	73.6	40.6	68.1	68.1	33.7	100.0	100.0
cholesky	68.3	97.3	99.6	99.6	95.6	99.2	95.6
correlation	5.1	23.9	54.8	60.4	60.4	99.4	98.8
covariance	4.8	23.3	58.9	73.0	38.1	57.5	100.0
doitgen	46.6	48.5	32.0	67.7	66.6	100.0	31.7
durbin	70.5	98.1	98.7	99.4	98.6	98.7	97.9
dynprog	59.2	69.0	83.8	84.5	77.6	81.3	77.6
fdtd-2d	50.6	41.4	98.7	98.7	17.5	62.8	55.0
fdtd-apml	70.0	94.7	97.9	97.9	100.0	100.0	99.3
floyd-warshall	71.3	25.1	62.9	74.9	100.0	100.0	100.0
gemm	11.4	32.9	98.2	100.0	69.1	83.5	96.9
gemver	28.8	52.4	65.7	100.0	96.7	98.6	65.7
gesummv	73.7	66.7	66.8	67.0	100.0	100.0	83.7
gramschmidt	9.3	20.2	44.3	44.3	13.3	43.7	43.5
jacobi-1d-imper	77.2	81.9	93.8	94.4	94.7	99.1	99.1
jacobi-2d-imper	49.2	61.9	80.3	80.6	98.8	98.9	68.2
lu	65.7	21.8	78.5	79.0	41.5	41.5	100.0
ludcmp	69.4	94.3	83.6	99.6	95.9	99.6	86.0
mvt	31.9	66.8	98.0	98.0	90.8	99.6	53.3
reg_detect	70.4	89.3	86.7	99.7	98.2	98.2	98.2
seidel-2d	42.4	54.7	49.6	50.2	61.4	61.4	100.0
symm	70.0	94.1	98.4	98.5	97.3	98.3	98.5
syr2k	35.4	46.8	51.3	52.1	51.5	84.6	90.0
syrk	19.2	32.9	41.4	44.8	27.8	90.5	89.5
trisolv	80.8	44.8	57.3	57.7	62.6	100.0	100.0
trmm	25.2	32.2	57.4	58.5	35.9	100.0	90.6
Average	47.9	54.5	73.4	79.4	70.2	87.4	86.1

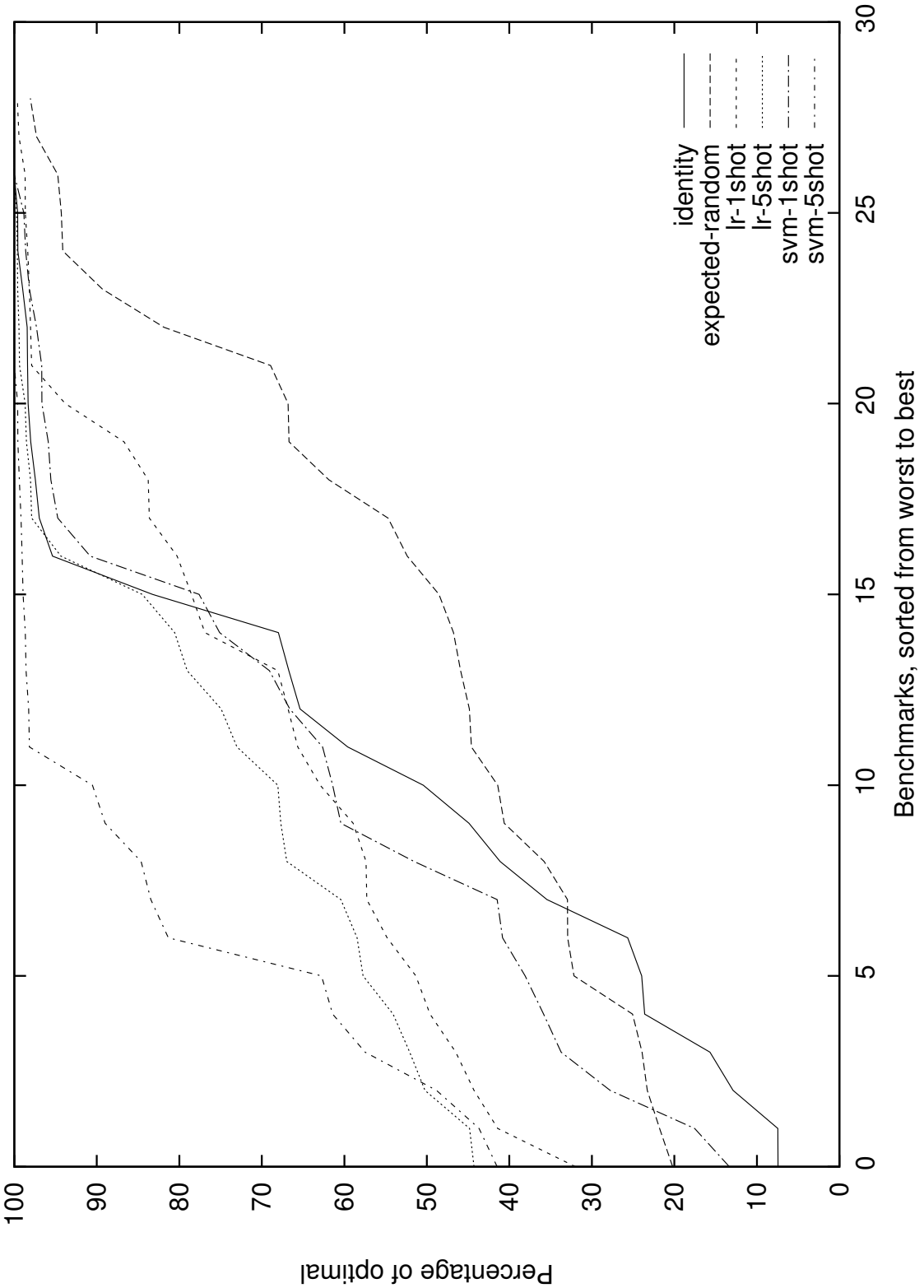


Figure 5.1: Percentage of optimal, Linear Regression and SVM compared to the baseline

Figure 5.2: Percentage of optimal, Decision Tree Vote compared to the baseline

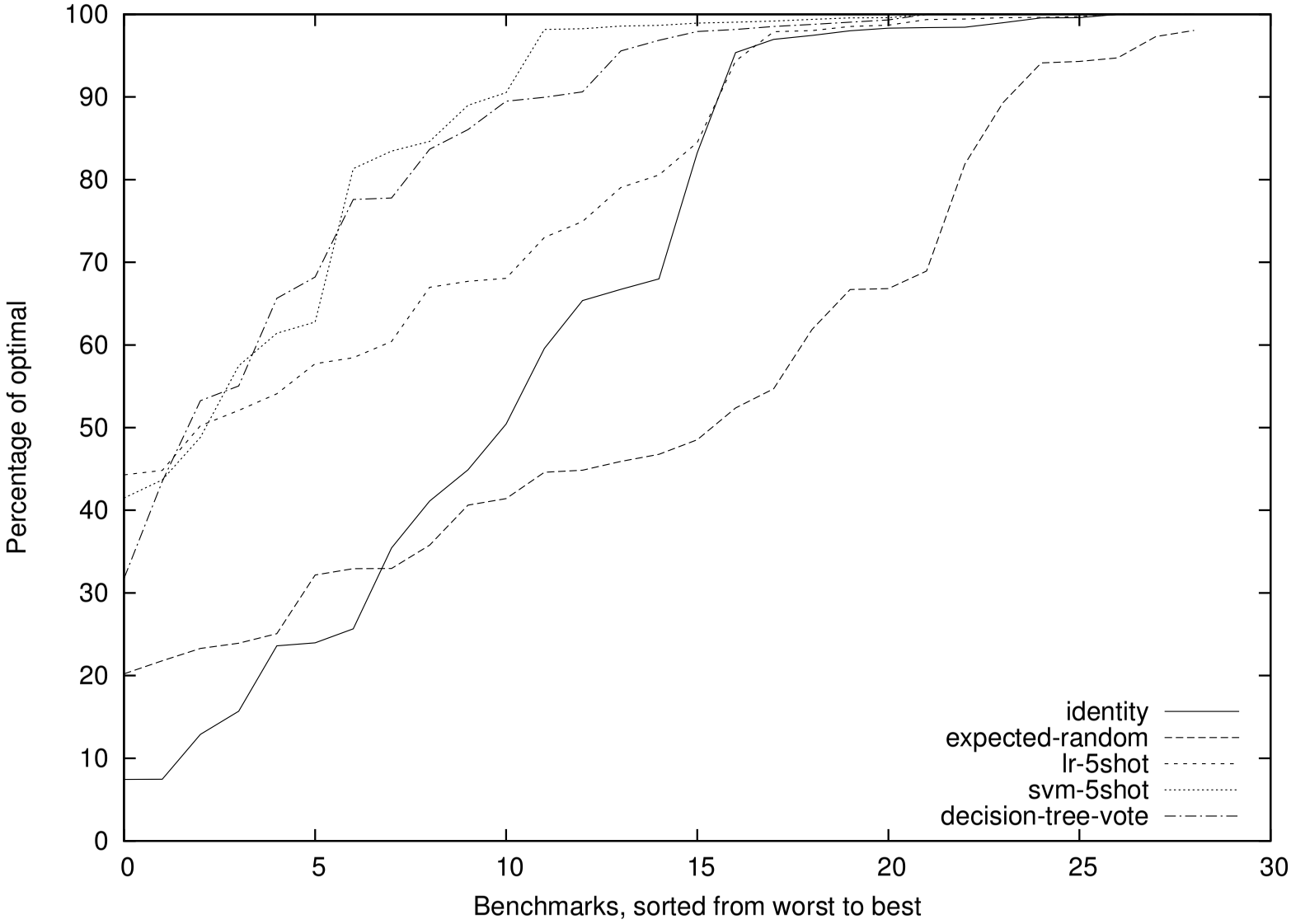


Table 5.3: Linear Regression, SVM, and Decision Tree Vote: Speed-up over Identity Transformation

Benchmark	Expected	Linear Regression		Support Vector Machine		Decision
	Random	1-shot	5-shot	1-shot	5-shot	Tree Vote
2mm	2.80	4.83	6.29	6.08	6.29	4.89
3mm	2.41	5.22	5.22	3.95	4.68	5.25
adi	0.49	0.63	0.74	0.56	0.67	1.36
bicg	0.55	0.92	0.92	0.46	1.36	1.36
cholesky	1.43	1.46	1.46	1.40	1.45	1.40
correlation	4.67	10.71	11.80	11.80	19.42	19.30
covariance	4.81	12.17	15.08	7.87	11.88	20.66
doitgen	1.04	0.69	1.45	1.43	2.15	0.68
durbin	1.39	1.40	1.41	1.40	1.40	1.39
dynprog	1.16	1.42	1.43	1.31	1.37	1.31
fdtd-2d	0.82	1.95	1.95	0.35	1.24	1.09
fdtd-apml	1.35	1.40	1.40	1.43	1.43	1.42
floyd-warshall	0.35	0.88	1.05	1.40	1.40	1.40
gemm	2.90	8.63	8.79	6.07	7.34	8.51
gemver	1.82	2.28	3.47	3.36	3.42	2.28
gesummv	0.90	0.91	0.91	1.36	1.36	1.13
gramschmidt	2.17	4.76	4.76	1.43	4.69	4.67
jacobi-1d-imper	1.06	1.22	1.22	1.23	1.28	1.28
jacobi-2d-imper	1.26	1.63	1.64	2.01	2.01	1.39
lu	0.33	1.20	1.20	0.63	0.63	1.52
ludcmp	1.36	1.21	1.44	1.38	1.44	1.24
mvt	2.09	3.07	3.07	2.84	3.12	1.67
reg_detect	1.27	1.23	1.42	1.39	1.39	1.39
seidel-2d	1.29	1.17	1.18	1.45	1.45	2.36
symm	1.34	1.40	1.41	1.39	1.40	1.41
syr2k	1.32	1.45	1.47	1.46	2.39	2.54
syrk	1.72	2.16	2.34	1.45	4.72	4.67
trisolv	0.55	0.71	0.71	0.77	1.24	1.24
trmm	1.28	2.28	2.32	1.42	3.97	3.60
Average	1.58	2.72	3.02	2.38	3.33	3.53

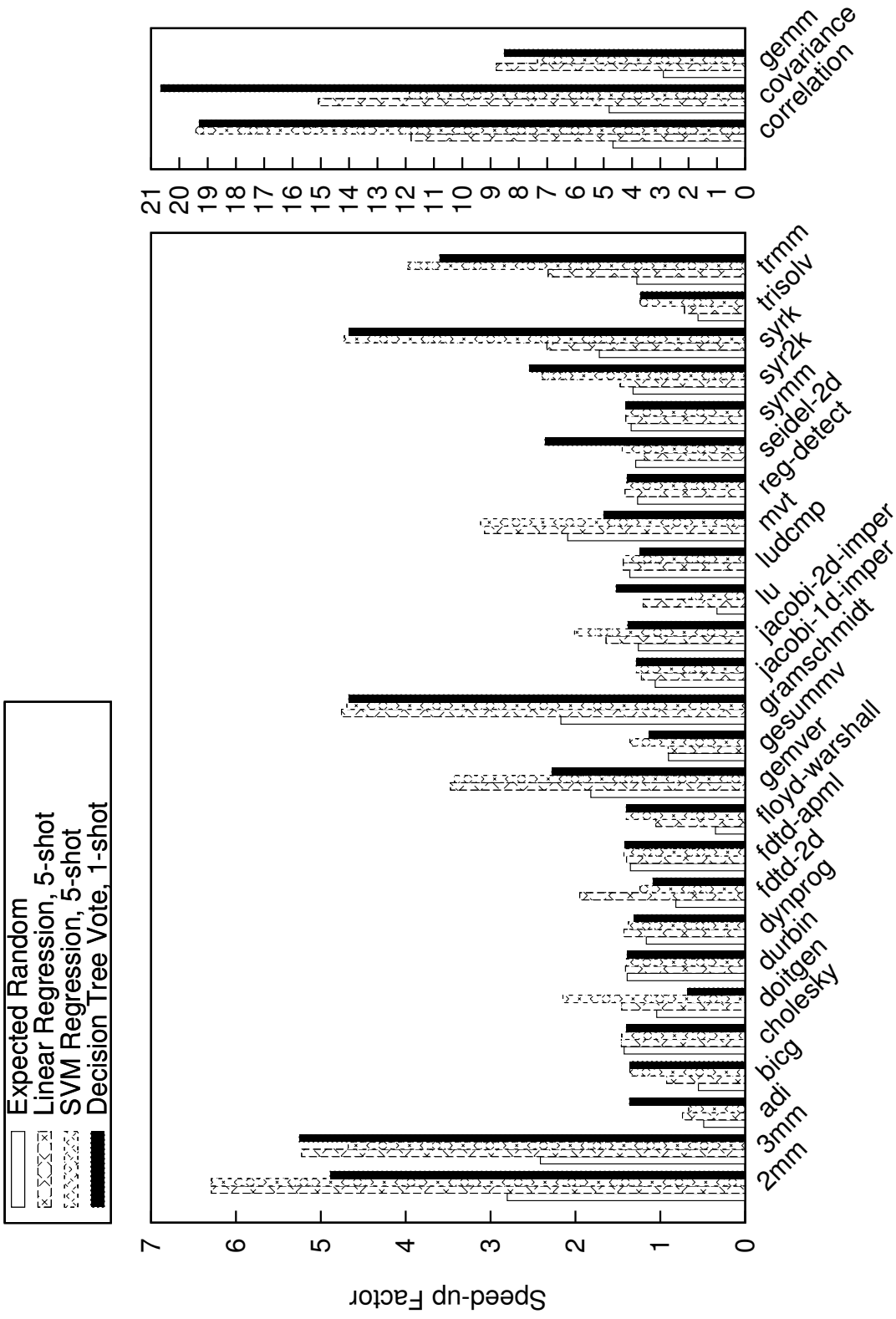


Figure 5.3: Speed-up Over Identity Transformation

5.5 Summary

In this chapter, I presented the results of the decision tree voting approach described in Chapter 4 and compared them to previous state-of-the-art results first presented by Park et al. [23]. The results of the decision tree approach rival those of the best approach presented by Park, five-shot SVM, in terms of efficacy at selecting a transformation sequence yielding as close to the best runtime as possible, while at the same time the decision tree voting approach does not require the execution of five binaries to reach that level of efficacy. This makes decision tree voting more usable in practice, where the kernel being optimized can be computationally intensive and running it multiple times can be time consuming.

Chapter 6

Conclusion

In this chapter, I summarize the highlights of the thesis and discuss some potential avenues for future work.

6.1 Summary

In recent years, the increasing complexity of CPU pipelines, the decrease in clock speed growth rates, and the increase in core count have combined to make it more difficult to write performant computational kernels. An automatic optimization mechanism for computational kernel loop nests would ease the task of software developers by allowing them to focus on domain-specific considerations and not have to worry about architectural concerns. The optimization mechanism should operate on straightforwardly-written loops and transform the loops such that they most efficiently utilize the features and constraints of the architecture where the program will run, while preserving the semantics of the original source. The Polyhedral Model [10] allows for the generation of semantic-preserving loop transformations, varying parameters pertaining to loop tiling, fusion, unrolling vecotirzation, and parallelism. Several approaches have been proposed for selecting a transformation among the sequence that can be generated by applying the Polyhedral Model. Some [4, 5] are model-driven and construct an analytic cost function which is then minimized, while others [27, 28] make use of the analytic approach where it has proven to work well while also employing an empirical (trial-and-error) methodology in areas where the model is found lacking. Finally, Park et al. [23] use machine learning to select a transformation using hardware performance counters as features. Park et al. explore two machine learning approaches: linear regression and SVM. In both cases, they attempt to predict the runtime improvement of a program that has had a particular transformation applied over the program with no transformations applied. They find that SVM performs better than linear regression, and that regardless of the machine learning algorithm used, best results are achieved when the top five transformations recommended by the algorithm are tried out in practice by running the program with the transformations applied and taking the transformation corresponding to the best runtime of the five. Needing to run the program

five times can be exceedingly time-consuming, depending on the complexity of the program and the input.

In the thesis, I built on the work by Park et al. and presented an alternative machine learning approach that achieves results as good as their best results while removing the need to run the program being optimized five times. With the transformations in the search space ranked in reverse runtime order, the transformation selected by my classifier was, on average, in the 86th percentile. Several alterations were made to the approach by Park et al. First, the problem was expressed as a binary classification problem rather than a regression problem. Rather than predicting the runtime improvement of a particular transformation, a continous-valued class, I predict which one of two particular transformations has a faster runtime, a binary class. Second, the hardware performance counter features are augmented with features derived from the program source code structure. Third, the continuous feature values are discretized. Fourth, feature selection is performed. Decision trees are constructed for pairs of features, and the decision trees are evaluated on how effective they are at predicting transformation utility. The top 20 trees thus selected then vote on the class corresponding to each set of attribute values, with the attributes values encapsulating the discretized hardware performance counter values for the unmodified program and the transformation. The transformations are then ordered in descending order according to how many times a particular transformation is predicted to be better than any other transformation. The top transformation in the list is then taken to be the best transformation in the search space.

I evaluated my proposal on a benchmark suite selected to be representative of the range of computational kernels. A leave-one-out approach was utilized both at the feature selection stage and at the machine learning stage. The transformation selected by my algorithm was compared to the transformations chosen by the four algorithms presented by Park et al. In addition, two baseline values were also considered: the identity transformation (the original program unchanged) and a transformation selected at random. On average, the transformation selected by my approach was as good as the transformation selected by the most promising of Park et al.'s approaches, five-shot SVM. However, my approach did not require running the program to be optimized five times, making the optimization process less time consuming.

6.2 Future Work

In this thesis, the hardware performance counter values used for features were continuous. The values were then manually discretized by plotting them in increasing order and looking for patterns. Automatic discretization mechanisms should be explored.

Different approaches to feature selection can also be explored. When selecting the decision trees to participate in the final vote, currently the trees are evaluated based on the number of benchmarks for which the transformation suggested by the tree does better than the baseline. However, the magnitude of the improvement over the baseline is not taken into consideration. A tree that selects a transformation that is 1% better than the

baseline is considered equivalent to a tree that selects a transformation 50% better than the baseline. The ranking of the trees could be weighted relative to the improvement over the baseline.

The voting process could also be altered. Currently, the trees that are selected each have one vote. Each tree predicts the class by examining the majority value at the leaf node. Leaf nodes may contain more than one value because the decision tree algorithm did not find it advantageous to branch further. Therefore, the class returned by the decision tree will be one both if the leaf node has six 1's and one 0 and if the leaf node has four 1's and three 0's. Instead, the tree can also report the probability for the class. For example, in the case of six 1's and one 0, the probability would be 0.86, while in the case of four 1's and three 0's, the probability would be 0.57. Then, each tree's vote would be weighted with the probability associated with the class it chooses. In the two example cases, the weights would be 0.86 and 0.57 for the two trees.

The classifier proposed in this thesis relies on dynamic features, namely, hardware performance counter values. These features encode both information about the target architecture for which the program is being optimized and information about the program itself. To extract these features, the program being optimized must be run once. This requirement may be a barrier to adoption as it lengthens the overall compilation cycle. One potential avenue to explore to make the compilation cycle less onerous is the decoupling of the architecture features from the program features. One could explore the creation of a short-running program that would only need to be run once per target architecture, rather than once for every program being optimized, and that would generate the architecture features in one pass. These architecture features could then be subsequently used in the optimization process in conjunction with static features extracted from the body of the program, similar to the static features extracted by Fursin et al. [14].

References

[1] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[2] Cdric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, A Group, and Inria Rocquencourt. Putting polyhedral loop transformations to work. In *In LCPC16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, 2003.

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, Paphos, Cyprus, March 2010. Springer-Verlag. Classement CORE : A, nombre de papiers acceptés : 16, soumis : 56.

[4] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[6] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Research report 94-24, LIP, ENS-Lyon, France, September 1994.

[8] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[9] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21:313–348, October 1992.

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. 21(6):389–420, December 1992.

[11] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, 1996. Springer-Verlag.

[12] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing - a VLIW approach to architecture, compilers, and tools.* Morgan Kaufmann, 2005.

[13] Johannes Fürnkranz and Eyke Hüllermeier. Pairwise preference learning and ranking. In Nada Lavrač, D. Gamberger, Hendrik Blockeel, and L. Todorovski, editors, *Proceedings of the 14th European Conference on Machine Learning (ECML-03)*, volume 2837 of *Lecture Notes in Artificial Intelligence*, pages 145–156, Cavtat, Croatia, 2003. Springer-Verlag.

[14] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, 2008. MILEPOST project (http://www.milepost.eu).

[15] Sylvain Girbal, Nicolas Vasilache, Cdric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl J. of Parallel Programming*, 34:2006, 2006.

[16] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[17] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2 edition, 2009.

[18] Eyke Hüllermeier, Johannes Fürnkranz, Weiwei Cheng, and Klaus Brinker. Label ranking by learning pairwise preferences. *Artif. Intell.*, 172(16-17):1897–1916, November 2008.

[19] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, UK, 1994. Springer-Verlag.

[20] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[21] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer Berlin / Heidelberg.

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

[23] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 119–129, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Louis-Noël Pouchet. *Iterative optimization in the polyhedral model*. PhD thesis, PhD thesis, University of Paris-Sud 11, Orsay, France, 2010.

[25] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multi-dimensional time. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 90–100, New York, NY, USA, 2008. ACM.

[26] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[27] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. *SIGPLAN Not.*, 46:549–562, January 2011.

[29] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Sys. Morgan Kaufmann, second edition, June 2005.

[31] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

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

[33] Ting-Fan Wu, Chih-Jen Lin, and Ruby C. Weng. Probability estimates for multi-class classification by pairwise coupling. *J. Mach. Learn. Res.*, 5:975–1005, December 2004.

[34] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM.