Predicting Unroll Factors Using Supervised Classification

Mark Stephenson and Saman Amarasinghe
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, Massachusetts
{mstephen,saman}@mit.edu

Abstract

Compilers base many critical decisions on abstracted architectural models. While recent research has shown that modeling is effective for some compiler problems, building accurate models requires a great deal of human time and effort. This paper describes how machine learning techniques can be leveraged to help compiler writers model complex systems. Because learning techniques can effectively make sense of high dimensional spaces, they can be a valuable tool for clarifying and discerning complex decision boundaries. In this work we focus on loop unrolling, a well-known optimization for exposing instruction level parallelism. Using the Open Research Compiler as a testbed, we demonstrate how one can use supervised learning techniques to determine the appropriateness of loop unrolling. We use more than 2,500 loops — drawn from 72 benchmarks — to train two different learning algorithms to predict unroll factors (i.e., the amount by which to unroll a loop) for any novel loop. The technique correctly predicts the unroll factor for 65% of the loops in our dataset, which leads to a 5% overall improvement for the SPEC 2000 benchmark suite (9% for the SPEC 2000 floating point benchmarks).

1. Introduction

With enough time and effort, system engineers can create models that accurately describe architectural components. For example, Yotov et al. show that compilers and runtime systems can rely on human-made models to make informed decisions [23]. Unfortunately, it is not always easy to model complex systems because many of the architectural components are inextricably tied together. It is also difficult to model the compiler passes with which a given optimization may interfere. For example, register allocators are often written to ignore important interactions with the instruction scheduler.

Creating a reliable model upon which to base decisions requires expert knowledge of the system and a huge amount of trial-and-error tuning. This paper experiments with applying machine learning techniques to the problem of heuristic tuning. Essentially, we aim to automatically create system models. Learning techniques can often find sense in high-dimensional spaces, and thus they can be effectively applied to compiler optimizations where the resulting performance is a function of several variables.

As a case study we apply two machine learning techniques to the problem of loop unrolling. Because loop unrolling indirectly affects so many aspects of system performance, it is difficult to model the appropriateness of the optimization. We show that *near neighbor* (NN) classification and *support vector machines* (SVM) work remarkably well for predicting unroll factors. Our best machine learning classifier can predict with 65% accuracy the optimal unroll factor, and the optimal, or second-best unroll factor 79% of the time. For the benchmarks that we evaluate, this means the classifier is within 7% of the optimal solution 79% of the time.

We evaluate the implications of improved unrolling decisions using the Open Research Compiler and an Itanium® 2 architecture. The Open Research Compiler uses two loop unrolling heuristics: one is used when software pipelining is disabled, and the other is used in conjunction with software pipelining to find schedules with fractional II. We used the aforementioned learning techniques to automatically create heuristics for both cases.

When software pipelining is disabled, our best classifier achieves a 5% speedup (over ORC's heuristic) for the SPEC 2000 benchmarks, and a 9% speedup for the floating point benchmarks in that suite. However, as is clear from the history of ORC releases, the system is tuned with software pipelining in mind. In fact, the release history shows that a great deal of time was spent optimizing the unroll heuristics for software pipelining: every major release employed a different unrolling heuristic (the current version is 205 lines worth of C++ code). Because of this effort, our results when software pipelining is enabled are less dramatic. Our sys-

tem is able to create heuristics from scratch that achieve a slight increase in performance (1% over ORC with software pipelining enabled for the SPEC benchmarks). While ORC's heuristic is the product of multiple years of human tuning, our machine-learned versions took seconds to create (once the training data had been collected). The results we present in this paper show that machine learning techniques can model systems as well as human designers, but with much less effort.

The paper is organized as follows. The next section briefly states the contributions of this research. Section 3 describes the advantages and disadvantages of loop unrolling; it lists some important factors that one should consider when trying to determine whether unrolling a given loop will be desirable. Section 4 discusses our approach and our infrastructure. Section 5 describes the learning techniques that we employ, while Section 6 describes experiments with multi-class classification. Section 7 describes experiments with finding the most informative characteristics of the loop unrolling problem. Section 8 discusses some potential issues with using machine learning for heuristic design. Section 9 relates our work to previous work, and we conclude in Section 10.

2. Contributions

The novel aspects of our research are summarized here:

- We use multi-class classification to improve compiler decisions. Many compiler decisions involve choosing between one of many options, not just making a binary choice. While other compiler researchers have employed learning techniques for binary problems, none to our knowledge have tried to solve harder multi-class problems.
- We show that near neighbor classification and support vector machines are viable methods for improving compiler decisions.
- We show how to use feature selection to identify the most salient features of a compiler problem.

We have also released the instrumentation library that we wrote and the raw loop data that we collected so other researchers can easily apply their own learning techniques. Please visit our website for information:

http://www.cag.csail.mit.edu/metaopt

3. Loop Unrolling

Loop unrolling is a well known transformation in which the loop body is replicated a number of times. Since the backward branch is needed only after executing the entire unrolled body, loop unrolling reduces overhead by decreasing the number of branch operations. This can be particularly important for architectures that have high branching overhead. However, loop unrolling is primarily used to enable other optimizations, many of which target the memory system. For example, unrolling creates multiple static memory instructions corresponding to dynamic executions of a single operation. After unrolling, these instructions can be rescheduled to exploit memory locality. If the loop accesses the same memory locations on consecutive iterations, many of these references can be eliminated altogether with scalar replacement. Another method to reduce memory traffic utilizes a wide memory bus to transfer multiple words with a single load or store operation. Unrolling is key to exposing adjacent memory references [6, 12] so that they can be merged into a single wide reference.

Arguably, the most important aspect of loop unrolling is its ability to expose instruction level parallelism (ILP) to the compiler. After unrolling, the compiler can reschedule the operations in the unrolled body to achieve overlap among iterations. Such a scheme was first used in the Bulldog compiler [9] and is still important in compiling for machines that support a high degree of ILP. Typically, unrolling is combined with other transformations that increase the size of the scheduling window. Examples include trace scheduling [9] and hyperblock formation [14]. These techniques are particularly useful in scheduling for loops that contain control flow or function calls because of the difficulty these problems present to software pipelining.

Loop unrolling is an interesting optimization because it indirectly affects many aspects of system performance: the efficacy of the instruction scheduler, the software pipeliner, the register architecture, and the memory system are all influenced by loop unrolling. Because its impact is mainly noticed in secondary effects, it is difficult to decide when loop unrolling is appropriate. Superficially, loop unrolling appears to be an optimization that is always beneficial. However, loop unrolling can impair performance in many cases. The following non-exhaustive list considers some possible drawbacks to loop unrolling:

- The most acknowledged detriment of unrolling is that code expansion can degrade the performance of the instruction cache.
- Added scheduling freedom can result in an increase in the live ranges of variables, resulting in additional register pressure. Since memory spills and reloads are

typically long latency operations, this can negate the benefits of unrolling.

- Control flow also complicates unrolling decisions. If the compiler cannot determine that a loop may take an early exit, it will actually have to add control flow to the unrolled loop which may negate — or at the very least neutralize — the benefits of unrolling.
- Some compilers aggressively speculate on memory accesses. Execution time will increase if the scheduler chooses to speculatively hoist unrolled memory accesses that dynamically conflict.

Compilers and architectures are complex systems. The scheduler, the register allocator, and the underlying architecture interact in non-trivial ways; loop unrolling increases the aggressiveness of certain optimizations, which depending on the circumstances, may adversely affect other important optimizations and reduce overall performance. The only way to truly know what will work is to empirically evaluate decisions, because even human-designed models must be evaluated at some point to determine their effectiveness. It is the goal of this research to use empirical observations to train a learning algorithm how to make informed decisions.

4. Methodology and Infrastructure

This section briefly introduces supervised learning in terms of loop unrolling. A discussion of the infrastructure that we use to perform the experiments in this paper follows.

4.1. Our Approach: Supervised Learning

Supervised learning is performed on a set of *training examples*. Each training example $\langle \mathbf{x}_i, y_i \rangle$ is composed of a *feature vector* $\mathbf{x}_i$ and a corresponding *label* y_i. The feature vector contains measurable characteristics of the object under consideration. Training a classifier usually involves finding a mapping from feature vectors to output labels so that the overall classification error is minimized on the training examples. The hope is that an adequately trained classifier will also be able to accurately discriminate novel examples (examples that were not in the training set).

In our experiments, the feature vector contains loop characteristics such as the trip count of the loop, the number of operations in the loop body, the programming language the loop is written in, etc. We extract a feature vector for every unrollable loop in our suite of benchmarks. Table 1 shows a subset of the features that we extracted for the experiments in this paper. We collected 38 features for these experiments, but as we discuss later, using a well chosen subset of features improves classification accuracy.

Feature
The loop nest level.
The number of ops. in loop body.
The number of floating point ops. in loop body.
The number of branches in loop body.
The number of memory ops. in loop body.
The number of operands in loop body.
The number of implicit instructions in loop body.
The number of unique predicates in loop body.
The estimated latency of the critical path of loop.
The estimated cycle length of loop body.
The language (C or Fortran).
The number of parallel "computations" in loop.
The max. dependence height of computations.
The max. height of memory dependencies of computations.
The max. height of control dependencies of computations.
The average dependence height of computations.
The number of indirect references in loop body.
The min. memory-to-memory loop-carried dependence.
The number of memory-to-memory dependencies.
The tripcount of the loop (-1 if unknown).
The number of uses in the loop.
The number of defs. in the loop.

Table 1. A subset of features used for loop classification. These characteristics are used to train the classifiers.

In addition to the feature vector, we also extract a training label for each unrollable loop in our benchmark suite. The training label indicates which (mutually exclusive) optimization is the best for each training example. For the experiments presented in this paper, labeling the data is relatively straightforward; we measure each loop using eight different unroll factors ($1, 2, \ldots, 8$), and the label for the loop is the unroll factor that yields the best performance. Thus, for each example loop we have a vector of characteristics that describes the loop, and a label that indicates what the empirically found best action for the loop is. The task of a classifier is to learn how best to map loop characteristics ($\mathbf{x}_i$) to the observed labels (y_i) using all the examples in the training set.

While supervised learning is trained offline, the learned classifier can easily be incorporated into a compiler.

4.2. Computing the Accuracy

The accuracy numbers presented in this paper were computed using a methodology known as leave-one-out cross-validation (LOOCV) [8]. The approach allows machine learning researchers to estimate the *generalization* ability of a learning algorithm (i.e., how well new examples can be classified). LOOCV is an iterative process that iterates N times, where N is the size of the training dataset. On each iteration i, the technique removes the i^{th} example from the training set, trains the classifier using the remaining $N - 1$ examples, and then sees how well the resulting classifier categorizes the left-out example. The generalization accuracy is then the number of correctly classified left-out examples divided by the total size of the training set.

There are other methods available for estimating a classifier's accuracy, but LOOCV is particularly appealing when the size of the training set is small — which ours is — because the learning algorithm can be trained using nearly all the examples in the dataset.

4.3. Compiler and Platform

We used the Open Research Compiler (ORC v2.1) [18]— an open source research compiler that targets Itanium architectures— to evaluate the benefits of applying learning to loop unrolling. ORC is a well-engineered compiler whose performance rivals commercial compilers. The experiments in this paper target a 1.3 GHz Itanium 2 server running Red Hat Linux Advanced Server 2.1. We use `-O3` optimizations for all experiments in the paper. For the first set of experiments we disable software pipelining to strictly focus on the loop unrolling heuristic, but the second set of experiments enables all optimizations. In all cases we set the maximum unroll factor to eight. Unroll factors beyond eight do not compile properly for many of the loops in our training benchmarks.

4.4. Loop Instrumentation

Because this paper is concerned with loop optimizations, we instrumented ORC to measure the runtime of all innermost loops. The instrumented code assigns a counter to every loop in the program. Immediately before execution reaches an innermost loop, the instrumentation code captures the processor's cycle counter and places it in the loop's associated counter. When the loop exits, the cycle counter is again captured and the total running time of the loop is computed.

We invested much engineering effort minimizing the impact that the instrumentation code has on the execution of the program. We initially inserted procedure calls to an instrumentation library that started and stopped the loop timers. This methodology proved to be extremely intrusive since the caller-saved register allocator spilled many variables on each call to the instrumentation library.

Our current loop instrumentor inserts assembly instructions that start and stop the loop timers. This lightweight model allows the instruction scheduler to bundle instrumentation code with a loop's prologue and epilogue code. Furthermore, the instrumentor does not significantly impact register usage.

At all exit points in the program a call is made to our instrumentation library to print the cumulative running time of each loop in the program. This data is used to train the offline learning algorithms we use; the learning algorithm needs to know which loop optimization strategy is most beneficial for each loop, and thus these cycle counts form the basis of our labeled training dataset.

We realize that we cannot possibly measure loop runtimes without affecting the execution in some way. However, the fact that we were able to effectively train a learning algorithm using data collected by the instrumentation library is evidence that the impact of our measurements is minimal. Nevertheless, to further mitigate noise introduced by instrumentation, we only use loops that are run for at least 50,000 cycles. For instance, were we to train with loops that are only run for a few thousand cycles, a loop that sits on the edge of an instruction cache boundary could introduce huge amounts of noise; a cache miss would comprise a significant portion of the total runtime of the loop.

We run each benchmark 30 times for all unroll factors up to eight; an unroll factor of one corresponds to leaving the loop intact (rolled). For each loop we base the performance on the median runtime for each unroll factor.

4.5. Effort Involved

This section discusses the effort that was involved with the experiments presented in this paper. The instrumentation of ORC — which at the time was unfamiliar to us — was the most demanding task, taking about two weeks of intensive work. Collecting the labels was somewhat time consuming since we ran each benchmark 30 times for all unroll factors, but this step was completely unsupervised and only took a little longer than a week. Finally, when we had our training dataset, we prototyped several popular learning algorithms in Matlab, many of which are publicly available online.

Now that our infrastructure is in place, quickly retuning the unrolling heuristic to match architectural changes will be trivial. We will simply have to collect a new labeled dataset, which is a fully automated process, and then we can apply the learning algorithm of our choice. Contrast this with the tedious, manual retuning efforts currently employed today. Furthermore, we are in the position to create heuristics for other loop optimizations such as loop tiling and strip mining.

4.6. Benchmarks Used

We extracted training examples from 72 benchmarks taken from a variety of benchmark suites. We use benchmarks from SPEC 2000[1], SPEC '95, and SPEC '92 [21]. For SPEC benchmarks such as swim, where the application appears in two different SPEC suites, we use the newest

[1] Please note that we have excluded two SPEC 2000 benchmarks: We cannot compile 252.eon because it is a C++ program, and 191.fma3d does not compile correctly with our instrumentation library (it creates a different number of loops depending on the unroll factor, and thus features and labels cannot be correlated).

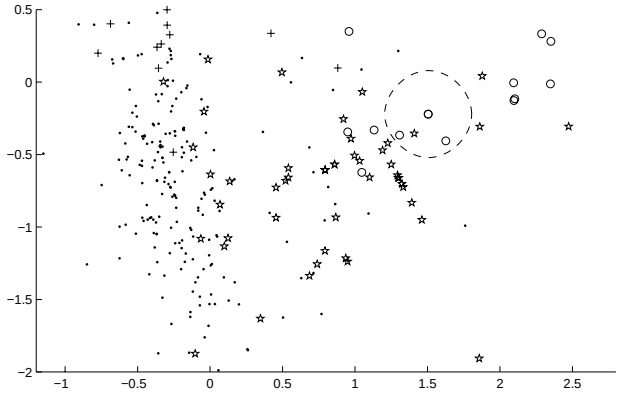

Figure 1. Near neighbor classification. This figure depicts the near neighbors algorithm on real unroll data. Note that this graphic is only meant to illustrate the idea of near neighbors; this figure only considers four classes, whereas the remainder of the paper considers classification into eight classes. To further improve visualization, we only include examples where the given unroll factor is at least 30% better than the other three.

version only. In addition, we train with Mediabench applications, benchmarks from the Perfect suite, and a handful of kernels. The training benchmarks span three languages (C, Fortran, and Fortran90). For each benchmark we only use loops that ORC can unroll and whose optimal unroll factor is measurably better than the average (1.05x) over all unroll factors up to eight.

There are many different classification techniques that one could choose to employ. The next section describes two techniques that work well for a wide range of problems.

5. Multi-Class Classification

This section describes two multi-class classification algorithms, which to our knowledge, we are the first to employ for compiler heuristic design. We begin by describing near neighbor classification, a conceptually simple, but highly effective technique. We then describe support vector machines, a statistical learning algorithm that is widely used in the machine learning community.

5.1. Near Neighbor Classification

Near neighbor (NN) classification is an extremely intuitive learning technique [8]. The idea of the algorithm is to construct a database of all $\langle \mathbf{x}_i, y_i \rangle$ pairs in the training set. A label (unroll factor) can be computed for a novel example simply by inspecting the labels of the nearest examples in the database. This is a sensible approach for assigning loop unroll factors: the compiler should treat similar loops similarly. We use Euclidean distance as the similarity metric. The distance between database entry $\mathbf{x_i}$ and a novel loop with feature vector $\mathbf{x_{novel}}$ is $\|\mathbf{x_{novel}} - \mathbf{x_i}\|$. The feature vector is normalized to weigh all features equally; otherwise, features with large values such as loop tripcount would grossly outweigh small-valued features in the distance calculation.

The graph in Figure 1 visually depicts the operation of NN on real loop data. Each of the points in the figure represents a loop from our suite of benchmarks. Points represented by pluses, circles, stars, and dots correspond to unroll factors one, two, four, and eight respectively. Because there are too many dimensions in the original feature space to graphically depict (equivalent to the number of features in Table 1), we have reduced the dimensionality by projecting loops from the original feature space — each of which is represented by a feature vector ($\mathbf{x_i}$) — onto a plane[2].

The near neighbors algorithm makes predictions for a new point based on the labels of points that lie within a specified radius of the new point. For all NN experiments we use a radius of 0.3, the value of which was determined experimentally. In Figure 1, the query point centered by the dotted circle has three neighbors that lie within the specified radius. The algorithm predicts that the unroll factor for the query point is the same label as the most commonly occurring label among the near neighbors. In this case, the algorithm would predict an unroll factor of two, represented by circles in the figure.

Near neighbors can be used to assign a confidence to a query. If the vast majority of near neighbors share the same label, then the confidence of the query is high. Alternatively, there are cases when there is no clear winner — or even no near neighbors — which corresponds to a low confidence. In these cases, we simply assign the unroll factor based on the label of the single nearest neighbor, but more elaborate schemes are certainly possible. One can imagine a tool that automatically detects outliers by setting low confidence examples aside. An engineer could then visually inspect outlier loops to determine why they are hard to classify.

Note that NN classification is trivial to train: one simply has to populate a 'database' of examples. Though the training time of a classifier is not a paramount concern (since training the classifier is done offline), the time it takes for the resulting classifier to make predictions is important (since this task will be performed by the compiler at compile time). NN classifies a new example by performing a linear scan of the examples in the training set. For small training sets like ours, the lookup is extremely fast: with over 2,500 examples in our database, the linear-time scan takes

[2] To find a 'good' plane onto which to project the data, we use the linear discriminant analysis algorithm described in [8]. Note that the axes of the graph correspond to a linear combination of the dimensions in the original feature space.

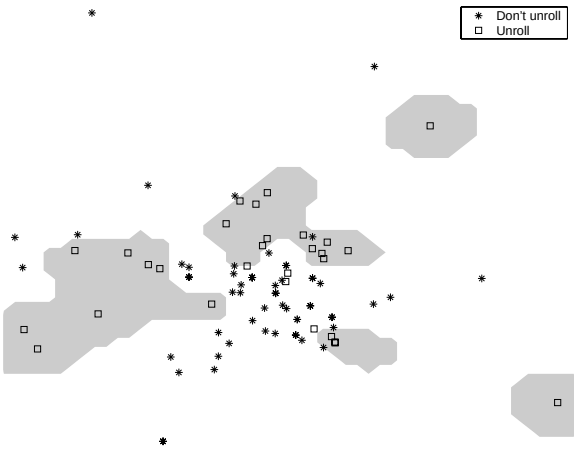

Figure 2. Support vector machine classification. This figure shows the classification of loop data by an SVM. In this example the SVM non-linearly maps the 2-dimensional feature space into a higher dimensional space (using a radial basis kernel function [8]). The SVM then finds the boundaries in the high dimensional space that maximally separate data from distinct classes. To improve visualization in this example we cast the original feature space to a 2-dimensional plane, we only consider binary classification, and we only consider examples where there is a 30% performance improvement.

less than 5 ms. Lookup time is far outweighed by compiler fixed-point dataflow analyses. Furthermore, advances in the area of approximate near neighbor lookup permit fast access (sublinear in the size of the database) to databases on the order of hundreds of thousands of examples, so we expect the NN method to scale well with database size [10].

5.2. Support Vector Machines

A detailed description of support vector machines (SVMs) is beyond the scope of this paper, so only the high level ideas of the algorithm are described here. The operation of an SVM is shown in Figure 2. There are two unique aspects of SVMs: first, an SVM maps the original D-dimensional feature space (using a non-linear function) to a higher-dimensional space where it is easier to 'separate' data, and second, in this transformed space the SVM attempts to find boundaries that maximally separate the classes. The latter aspect means that an SVM does not necessarily try to minimize the errors on the training set. Proponents of SVMs claim that this prevents 'overfitting' the training data, and thus will more likely generalize better to novel examples.

SVMs are binary classifiers, and thus some work must be done to use them in a multi-class classification context.

Prediction Correctness	NN	SVM	ORC	Cost
Optimal unroll factor	0.62	0.65	0.16	1x
Second-best unroll factor	0.13	0.14	0.21	1.07x
Third-best unroll factor	0.09	0.06	0.21	1.15x
Fourth-best unroll factor	0.06	0.06	0.13	1.20x
Fifth-best unroll factor	0.03	0.02	0.16	1.31x
Sixth-best unroll factor	0.03	0.03	0.04	1.34x
Seventh-best unroll factor	0.02	0.02	0.05	1.65x
Worst unroll factor	0.02	0.02	0.04	1.77x

Table 2. Accuracy of predictions for the nearest neighbors algorithm, an SVM, and ORC's heuristic. This table shows the percentage of the predictions that each algorithm made that were optimal. In addition, the table shows the percentage of predictions made by each algorithm that were Nth best. The SVM predicts the optimal or nearly-optimal unroll factor 79% of the time. The *Cost* column shows the average runtime penalty for mispredicting (as compared to the optimal factor).

While there are many ways to do this, one common method uses *output codes* [7]. Output codes associate a unique binary code to each label. For example, one possible set of codewords for a three-class problem is,

class	h_1	h_2	h_3
1	1	0	0
2	0	1	0
3	0	0	1

Now, the problem has been transformed into many binary classification problems. In the case of the above example, we would train three binary classifiers, each of which would use the binary partition induced by the codewords. Thus, classifier h_1 would learn to discriminate examples in class one from examples in classes two and three. A query's "code" representation is formed by concatenating the binary classifier predictions. The multi-class prediction is the class label corresponding to the closest codeword (in hamming distance) to the query's code. Error correcting codewords can provide better results by using more bits than necessary to describe each label, but for simplicity we do not use such encodings.

SVMs take longer to train than the NN algorithm (around 30 seconds for our data), but once the classifier has been constructed, unroll factors for novel examples can be predicted quickly. For a good description of the operation of SVMs please see [8].

6. Experiments with Multi-Class Classification

In this section we describe the operation of a multi-class classifier for loop unrolling. More specifically, we attempt to classify loops into one of eight categories, corresponding

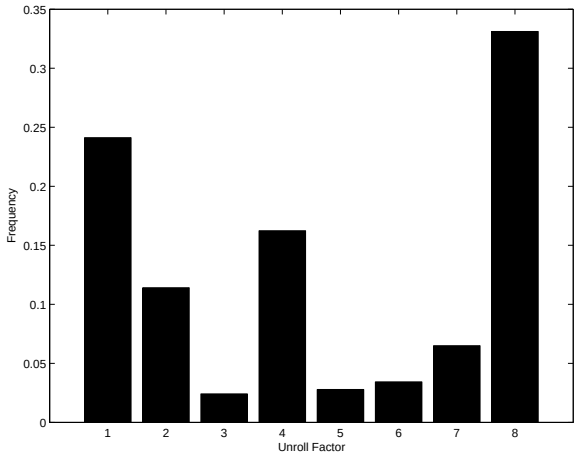

Figure 3. Histogram of optimal unroll factors. This figure shows the percentage of loops for which the given unroll factor is optimal. The histogram data was collected from over 2,500 loops (spanning several benchmarks suites) with software pipelining disabled.

to unroll factors one through eight. Recall that an unroll factor of one leaves the loop rolled.

As mentioned in Section 4, we first collect the amount of time it takes for each unroll factor to execute each unrollable loop in our suite of benchmarks. The unroll factor that requires the fewest number of cycles to execute a given loop is the label for that loop. We do not use the full set of 38 features that we extracted. Instead, as we will discuss in Section 7, we use the most "informative" subset of features for the classification experiments performed in this section.

We train the NN algorithm by simply populating the database with examples, and the predicted unroll factor for a novel loop will be the most common unroll factor of the loops within a radius of 0.3. Note that we chose this radius by inspecting the distances to training examples for several queries. For the SVM, we obtained the C and Matlab SVM implementation distributed at [13]. The toolkit contains functions for tuning, training, and testing the accuracy of an SVM.

Table 2 shows the accuracy of the learning algorithms and ORC's heuristic. The numbers in the table were collected with software pipelining disabled. Using leave-one-out cross validation we find that 65% of the time the SVM finds the optimal unroll factor. A further 14% of the time it chooses the nearly-optimal solution. The rightmost column in the table shows the cost associated with mispredicting. We can conclude from the table that a full 79% of the time, SVM classification is within 7% of the optimal performance (with this dataset). The NN algorithm performs slightly worse, achieving a 62% classification rate.

The histogram in Figure 3 shows the distribution of optimal unroll factors with software pipelining disabled. An interesting observation is that non-power of two unroll factors are rarely optimal for this dataset. The figure also indicates that no one loop unrolling factor is dominantly better than the others.

6.1. Realizing Speedups

In this section we see if improved unrolling classification accuracy yields program speedups. For these experiments, we compile the SPEC 2000 benchmarks using the learned classifiers to predict an unroll factor for each loop. Note that we train algorithms with all the examples in our training set minus the examples from the benchmark whose performance we are attempting to gauge. In other words, similar to LOOCV, when compiling a benchmark, we exclude all examples in that benchmark from the NN database. In this way we see how well the learned compiler algorithm performs on loops that it has not seen before. We do not instrument the compiled code for the experiments in this section. Instead we use the UNIX `time` command and the median of three trials to measure whole-program runtimes.

Figure 4 shows the performance improvement of our method over ORC's unrolling heuristic when software pipelining has been disabled. The figure also shows the speedup that the compiler could obtain if an "oracle" were to make its unrolling decisions. The SVM achieves a speedup on 19 of the 24 SPEC benchmarks. Overall our technique attains a 5% average speedup on the SPECs, and a 9% speedup when only the SPECfp benchmarks are considered. The oracle is slightly outperformed in a couple of cases because our data collection methodology is not perfect. In addition to working within a generally noisy environment, we assume that the optimal unroll factor of a particular loop does not depend on the unroll factors of the other loops. While this assumption may not be entirely correct, it simplifies the data collection process (we can collect the runtimes for all loops given a particular unroll factor in the same run). The overall performance of the oracle legitimizes our assumptions.

Figure 5 shows the performance of the predictors when software pipelining is enabled. Software pipelining exposes many of the benefits of loop unrolling, so in general loop unrolling will not yield the kinds of speedups seen in Figure 4. However, there are cases when unrolling will help the software pipeliner achieve a fractional initiation interval, thus improving performance. Likewise, too much unrolling may cause undo register pressure, impairing performance.

The NN classifier and the SVM outperform ORC's heuristic on 16 of the 24 predictors, leading to an overall improvement of slightly over 1%. Let us remind the reader

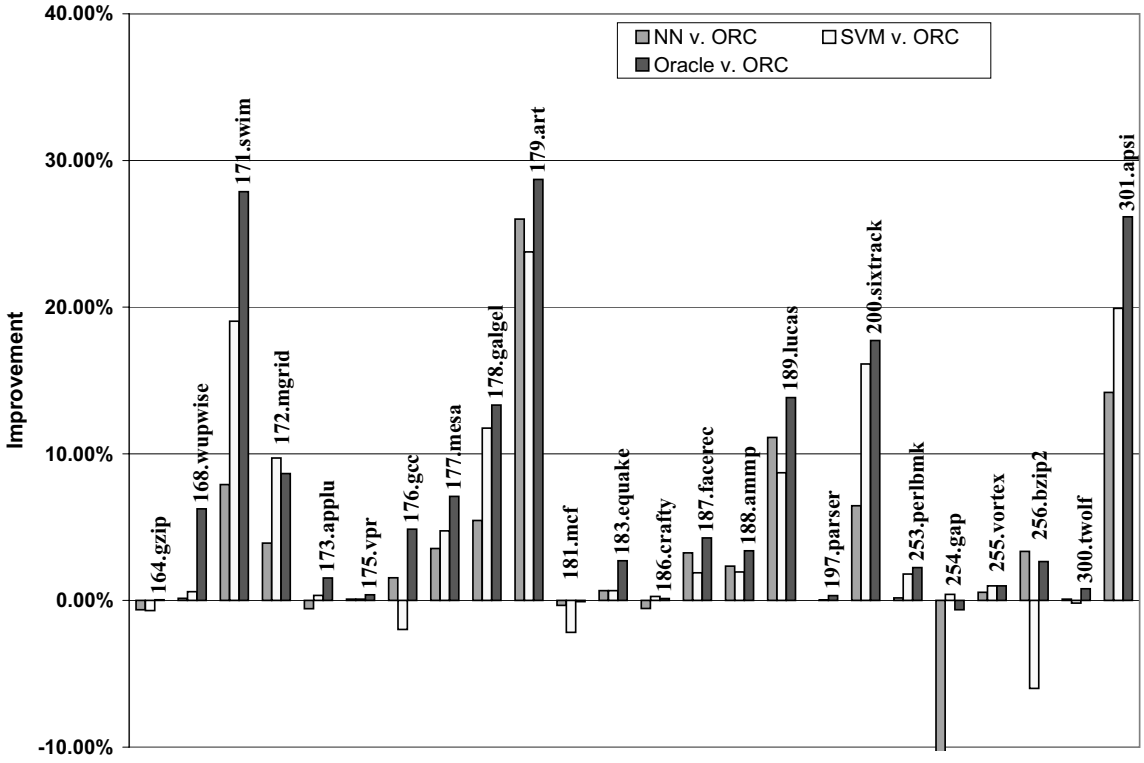

Figure 4. Realized performance on the SPEC 2000 benchmarks with SWP disabled. Both NN and an SVM achieve speedups on 19 of the 24 benchmarks. The SVM achieves a 5% speedup overall, and it boosts the performance of all SPECfp benchmarks, leading to a 9% overall improvement. Near neighbors performs slightly worse, boosting the performance by about 4%. The rightmost bar shows the speedup that an "oracle" would attain (7.2% average).

here that ORC is tuned with software pipelining in mind, and that every release of ORC to date has included a different unrolling heuristic. The current heuristic is around 200 lines of code. With that perspective in mind, the fact that machine learning algorithms can do the same task in a matter of seconds (days including the time it takes to collect the labels) is exciting.

Note that the training sets for 177.mesa, 181.mcf, and 186.crafty are obviously noisy since ORC's heuristic outperforms the oracle. Future work will consider techniques to reduce the amount of noise in the training datasets.

7. Feature Selection

This section focuses on finding the most informative features for discriminating unroll factors. We take two approaches to feature selection in this section. The first method uses information theory to score the information content of a feature. The second method greedily chooses features that match a given classifier for a given training set.

Rank	Feature	MIS
1	# floating point operations	0.19
2	# operands	0.186
3	instruction fan-in in DAG	0.175
4	live range size	0.16
5	# memory operations	0.148

Table 3. The best five features according to MIS.

7.1. Mutual Information Score

The *mutual information score* (MIS) measures the reduction in uncertainly in one variable (*e.g.*, a particular feature f) given information about another variable (*e.g.*, the best unroll factor u) [8]. The MIS adapted for our problem is given by,

$$I(f;u) = \sum_{\phi \in J} \sum_{y \in \{1...8\}} P(\phi, y) \cdot \log_2(\frac{P(\phi,y)}{P(y)P(\phi)}),$$

where J represents the set of values that f can assume, and for our problem u can assume values in $\{1, \ldots, 8\}$. We use the MIS to determine the extent to which knowing

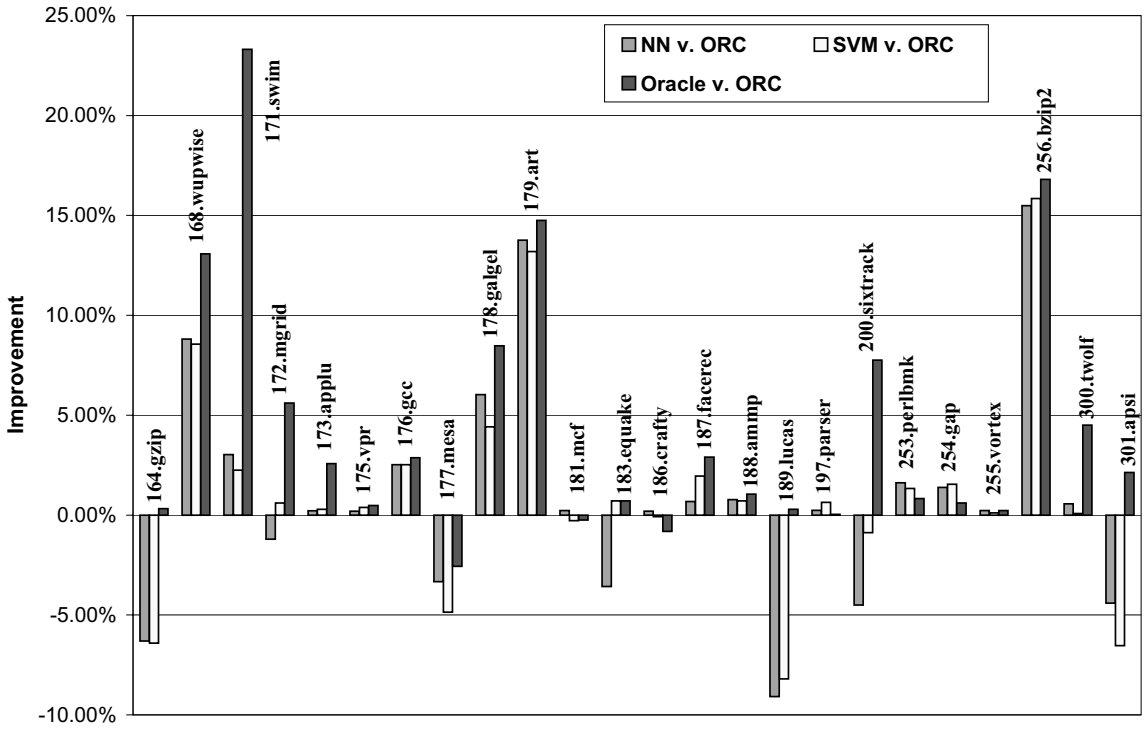

Figure 5. Realized performance on the SPEC 2000 benchmarks with SWP enabled. We attain speedups on 16 of the 24 benchmarks in this graph, and a 1% speedup overall. The rightmost bar for each benchmark shows the speedup that a 'perfect' classifier would attain (4.4% overall). Note that the training sets for 177.mesa, 181.mcf, and 186.crafty are noisy since the oracle is outperformed by ORC.

Rank	NN	Error	SVM	Error
1	# operands	0.48	# floating point operations	0.59
2	live range size	0.06	loop nest level	0.49
3	critical path length	0.03	# operands	0.34
4	# operations	0.02	# branches	0.20
5	known tripcount	0.02	# memory operations	0.13

Table 4. The top five features chosen by greedy feature selection for two different classifiers on our dataset. The error numbers reported here are for the training set, hence the low error rates for these classifiers.

the value of a loop characteristic reduces the uncertainty about the desired loop unroll factor. Informative features will receive higher scores than uninformative features. We bin the values of continuous features before estimating the probability mass functions used to compute the MIS.

Table 3 shows the five features with the highest MIS.

7.2. Greedy Feature Selection

There are several problems with MIS, most notably that it does not tell us anything about how features interact with each other. In addition, even though the score is a metric for information content, it does not guarantee that the features will be useful for a particular classifier.

Greedy feature selection identifies features that perform well for a given classifier and a given training dataset. Given a feature set, $F = \{f_0 \dots f_N\}$, the simple algorithm starts by choosing the *single* best feature, $b_0 \in F$, for discriminating the training dataset (using a particular classifier). The algorithm proceeds by choosing a second feature, $b_1 \in F$, that together with b_0, best discriminates the training dataset. In each iteration of the algorithm a new feature is chosen that minimizes the training error given the features that have already been selected. The algorithm halts after a user-defined number of features have been selected.

Table 4 shows the best five features for our dataset according to greedy feature selection. Notice that the choice of classifier affects the list of features deemed to be the most informative. For the NN algorithm, instead of looking for examples within a set radius, we modified the algorithm so that it looks for the single closest point in the database to the query and assigns a prediction accordingly.

We used the union of the features in Table 3 and Table 4 to perform the classification experiments presented in Sec-

tion 6. Whenever possible, it is preferable to use a small number of features when training a classifier. Uninformative features can 'confuse' a learning algorithm or lead to overfitting of the training data. In addition, learning algorithms are generally more efficient when shorter feature vectors are used.

Notice that the number of instructions in the loop body appears only once in Tables 3 and 4, and relatively far down the list. We highlight this fact because this feature is the de facto standard when discussing unrolling heuristics. According to both feature selection methods applied above, there are many other features that are more useful for discriminating unroll factors. The features that are listed in this section are not entirely surprising; arguments could be made to support their predictive values. It would be more difficult however, to determine why the features found by greedy selection are *jointly* informative.

8. Discussion

We believe that machine learning techniques have the potential to radically alter compiler construction methods. Future compilers may be designed in such a way that human designers can concentrate on the correctness of program transformations and optimizations, leaving the grunt work of heuristic selection and tuning to machine learning methods. However, there are still many issues that need to be addressed before such a vision becomes reality. In this section we describe some of the advantages and shortcomings of designing heuristics with machine learning.

One argument against using machine learning is that the compiler writer must extract the features with which the learning algorithms are trained. However, most of the features that we used in this paper were readily available from the ORC infrastructure. In the future, if these techniques are proven to be widely accepted, compiler passes will provide 'feature extraction' tools, much like compiler infrastructures provide generic data flow analysis packages.

Another potential reason for not using machine learning to optimize compiler heuristics is that extracting features and labels takes time. While the actual training of the classifiers takes less than a few seconds, it does take time to acquire the labels. However, collecting the labels was a completely unsupervised process, and in our opinion, required far less effort than constructing an unrolling heuristic by hand. Again, in the future, compiler infrastructures may export generalized timers to aid in label extraction.

In terms of the ability to model a system, learned heuristic predictions are confined to the limits of the labels with which they were trained (e.g., our learned classifiers will never predict unroll factors greater than eight). While this limitation may prevent the technique from being used for some compiler optimizations, the vast majority of optimizations already have imposed constraints. For our experiments, we set our limits to the greatest unroll factor for which all of the loops in our training set compile correctly. That said, future work will consider *regression*, which can predict values outside the range of the labels with which the learning algorithm is trained.

Finally noise presents a challenge to automatically learning compiler heuristics. The finer the granularity at which execution is measured, the noisier the measurements become. Modern architectures are helping our cause by including several user-readable performance counters. However, it will never be possible to eliminate noise in a multi-workload environment. Future research will explore ways to reduce noisy measurements.

9. Related Work

This section discusses relevant related work. Because our research focuses on applying learning techniques to compilation, we emphasize related work in this area.

Monsifrot et al. use a classifier based on "Boosted" decision tree learning to determine which loops to unroll [16]. While the methodology we present in this paper is similar, our work differs in several important ways. Whereas our experiments employ multi-class classification to determine the optimal unroll factor, their work only considers binary classification, leaving the choice of unroll factor up to a compiler heuristic. Doing so, their learned classifier correctly predicts 86% of the loops in their benchmark suite. Judging by the histogram in Figure 3, simply unrolling all the time will achieve 77% accuracy, and while unrolling may be better than not unrolling for a given example, Table 2 shows that choosing the wrong unroll factor can severely limit performance.

Calder et al. used neural networks and decision trees, both of which are supervised learning techniques, to fine-tune static branch prediction heuristics [1]. While their technique is effective, branch prediction is a binary problem that is simpler than the multi-class problem this paper considers. Finally, their problem has the benefit that instrumentation code to determine branch direction will not affect the direction to which branches are resolved. They were therefore able to work with a noiseless dataset. We must deal with noisy datasets; we measure execution time, but the instrumentation counters we insert have an effect on the measurement.

Cavazos and Moss use supervised learning to improve the compilation speed of a Java JIT compiler [4]. They train a learning algorithm to recognize when the compiler can forgo scheduling a basic block without sacrificing much performance. The resulting JIT compiles code faster while retaining 90% of the performance of scheduling every basic block. While their problem is interesting, again, it is a

binary problem.

Moss et al. [17] and McGovern et al. [15] focused on scheduling straight line code. Moss used machine learning to discover a preference for scheduling instructions in a ready worklist, and McGovern built upon that work. Because of the combinatorial blowup of measuring all permutations of instructions, it is unclear whether a supervised approach applies to this problem.

In previous work we used genetic programming to fine-tune compiler priority functions [22]. The reinforcement learning framework used for that work suited the problem well. However, supervised learning of the form presented in this paper is more efficient whenever a labeled training dataset can be created. Our reinforcement learning approach requires weeks to train, while most supervised learning algorithms require minutes or seconds (once the features and labels have been collected). In addition, genetic programming is a random process where back-to-back runs yield different results.

Cooper et al. [5], Puppin et al. [19], and Kulkarni et al. [11] use genetic algorithms to search for effective compiler phase orderings. Genetic algorithms are well-suited to their task, but genetic algorithms can be unstable and their fixed-length representation precludes their use in many problems.

Several compiler researchers have created model-based systems to automatically compute unroll factors [20, 3, 2]. In particular Sarkar [20] used in-depth, hand-made system models to create a cost function that ranks unroll factors according to estimated performance improvement. His technique improved a highly optimized, industry-strength compiler by 8% on seven of the SPEC95fp benchmarks. While our test infrastructures are different (and probably not comparable), it is worth noting that we achieved an 9% improvement on the SPECfp benchmarks.

10. Conclusions and Future Work

Compilers rely on models to make informed decisions. While humans can generate highly effective models, the number of person hours required to create them may be prohibitive. This research experimented with the automatic creation of compiler heuristics using supervised machine learning techniques. We used empirical evidence to teach a simple machine learning algorithm how to make informed loop unrolling decisions.

The learned classifiers predict loop unrolling factors with good precision. Using leave-one-out cross-validation to find the generalization ability of the classifier, the algorithm is able to predict the optimal unroll factor for a given loop 65% of the time. Furthermore it predicts the optimal, or the nearly optimal solution 79% of the time. We translate these results into speedups on a real machine. Using the Open Research Compiler and targeting the Itanium 2 architecture, we find that the learning algorithms improve the performance of 19 of the 24 benchmarks in the SPEC 2000 benchmark set. When we focus solely on loop unrolling we achieve a 5% improvement on the SPEC benchmarks, while improving the SPECfp benchmarks by 9%. With software pipelining enabled, the machine-learned heuristics slightly outperform ORC's heuristic.

In this research, apart from extracting features that we think *might* be pertinent, we purposefully thought little about designing an unrolling heuristic. Furthermore, almost no time went into tweaking the machine learning algorithms. Therefore, while the performance results are satisfying, we are more excited about the complexity ramifications of our research. We believe our method requires less effort than traditional trial-and-error heuristic tuning. And now that our infrastructure is in place, we are in the position to quickly improve many other loop optimizations (e.g., loop tiling, strip mining, hyperblock formation in loops, etc.), some of which future work will consider.

We believe that engineers and system modelers can benefit from machine learning tools that distill the most important characteristics of an optimization. We used feature selection to identify the most salient features for predicting unroll factors. Furthermore, we improved the prediction and runtime performance of our learning algorithms by using a reduced feature set size for classification. Our eventual goal is to distribute machine-learning-based tools that automatically identify the most important characteristics of a given optimization.

Compiler writers are forced to spend a large portion of their time designing heuristics. The results presented in this paper lead us to believe that machine-learning techniques can create certain heuristics well, and at the very least, can help point engineers in the right direction.

11. Acknowledgments

We thank the reviewers of this paper who provided us with excellent and detailed feedback. Thanks to Sam Larsen who helped write Section 3, and to Rodric Rabbah and Kristen Grauman who have read multiple drafts of this paper. Thanks to Una-May O'Reilly and Leslie Kaelbling for various helpful discussions. This research was partially supported by DARPA grant F29601-03-2-0065.

References

[1] B. Calder, D. G. ad Michael Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (ToPLaS-19)*, volume 19, 1997.

[2] S. Carr and Y. Guan. Unroll and Jam using Uniformly Generated Sets. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[3] S. Carr and K. Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. In *ACM Transactions on Programming Languages and Systems (ToPLaS-16)*, November 1994.

[4] J. Cavazos and E. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004*. ACM, 2004.

[5] K. Cooper, P. Scheilke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, pages 1–9, 1999.

[6] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 186–195, Orlando, FL, June 1994.

[7] T. Dietterich and G. Bakiri. Solving Multiclass Learning Problems via Error-Correcting Output Codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[8] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, 2001.

[9] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1985.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th Conference on Very Large Data Bases*, pages 518–529, February 1999.

[11] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *In Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*. ACM, 2003.

[12] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, June 2000.

[13] Least Squared Support Vector Machines (LSSVM). *http://www.esat.kuleuven.ac.be/sista/lssvmlab/*.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proc. 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, December 1992.

[15] A. McGovern and E. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *Proceedings of Neural Information Processing Symposium*, 1998.

[16] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 41–50, 2002.

[17] E. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovi, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Proceedings of Neural Information Processing Symposium*, 1997.

[18] Open Research Compiler. *http://ipf-orc.sourceforge.net*.

[19] D. Puppin, M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Adapting Convergent Scheduling Using Machine Learning. In *Proceedings of the '03 Workshop on Languages and Compilers for Parallel Computing*, College Station, TX, 2003.

[20] V. Sarkar. Optimized Unrolling of Nested Loops. In *Proceedings of the 14th Internatin Conference on Supercomputing*, Santa Fe, NM, 2000.

[21] SPEC.org. *http://www.spec.org*.

[22] M. Stephenson, M. Martin, U.-M. O'Reilly, and S. Amarasinghe. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[23] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, and K. Pingali. A comparison of empirical and model-driven optimization. In *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 63–76. ACM, 2003.