Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2004-012
MIT-LCS-TR-938

March 22, 2004

Predicting Unroll Factors Using Nearest Neighbors

Mark Stephenson and Saman Amarasinghe

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Predicting Unroll Factors Using Nearest Neighbors

Mark Stephenson and
Saman Amarasinghe
Massachusetts Institute of Technology
Computer Science and
Artificial Intelligence Laboratory
Cambridge, MA 02139
{mstephen, saman}@cag.csail.mit.edu

ABSTRACT

In order to deliver the promise of Moore's Law to the end user, compilers must make decisions that are intimately tied to a specific target architecture. As engineers add architectural features to increase performance, systems become harder to model, and thus, it becomes harder for a compiler to make effective decisions.

Machine-learning techniques may be able to help compiler writers model modern architectures. 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 our 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 model the appropriateness of loop unrolling.

We use more than 1,100 loops — drawn from 46 benchmarks — to train a simple learning algorithm to recognize when loop unrolling is advantageous. The resulting classifier can predict with 88% accuracy whether a novel loop (i.e., one that was not in the training set) benefits from loop unrolling. Furthermore, we can predict the optimal or nearly optimal unroll factor 74% of the time. We evaluate the ramifications of these prediction accuracies using the Open Research Compiler (ORC) and the Itanium® 2 architecture. The learned classifier yields a 6% speedup (over ORC's unrolling heuristic) for SPEC benchmarks, and a 7% speedup on the remainder of our benchmarks. Because the learning techniques we employ run very quickly, we were able to exhaustively determine the four most salient loop characteristics for determining when unrolling is beneficial.

1. INTRODUCTION

It is difficult to model modern computer architectures. Even earnest attempts to create accurate system-level models can yield poor results. There is a good reason for this: the components in modern architectures are inextricably tied together. Modern compilers are also extremely complicated tools. Compiler writers have broken the difficult problem of compilation into manageable phases that are solved in isolation. Due to the nature of the problem, it is not possible for a compiler writer to account for the interactions between compiler phases.

In addition, there are intricate interactions between the compiler and the underlying architecture. For instance, the configuration of the memory system affects the efficacy of the instruction scheduler, and vice versa, the instruction scheduler affects the performance of the memory system.

Compiler writers rely on simple localized models to abstract away system complexities. For example, register allocators are often written to ignore important interactions with the instruction scheduler and instead base decisions solely on simple models of the memory system. Without a reliable model upon which to base decisions, compiler writers necessarily resort to trial-and-error heuristic tweaking to achieve a suitable performance.

The goal of this research is to show that machine-learning techniques can be used to improve compiler heuristics. We train a compiler (with empirical data) to recognize when it is worthwhile to perform a particular optimization. In essence, we train the compiler to perform better. Learning techniques can find sense even in high-dimensional search spaces, and thus they can be effectively applied to compiler optimizations where the resulting performance is a function of several variables.

We apply a simple machine-learning technique to the problem of loop unrolling. We first try to determine when loop unrolling is beneficial. We show that *nearest neighbor* classification, a simple and widely used learning technique works remarkably well. A nearest neighbor classifier can predict with 88% accuracy when a loop should be unrolled.

We then extend this result by showing that we can train a nearest neighbor classifier to predict not only when a loop should be unrolled, but the factor by which it should be unrolled. When we consider all unroll factors through eight, we can predict the optimal unroll factor 60% of the time, and the optimal or nearly optimal factor 74% of the time. We evaluate the implications of improved unrolling decisions using the Open Research Compiler and an Itanium® 2 architecture. With this infrastructure, the nearest neighbor algorithm achieves a 6% speedup (over ORC's heuristic) for the SPEC benchmarks and a 7% speedup on an assortment of synthetic benchmarks and kernels.

While we are pleased with the performance results of our research, we are more excited about the prospects of using our work to reduce the complexity of compiler development; we show that it is possible to offload much compiler design to machine-learning algorithms. Apart from creating a *train-*

ing data set by which an offline (i.e., performed at compiler development time) learning algorithm can be trained, our technique requires very little tweaking on the part of the compiler writer. Furthermore, engineers can use learning techniques, as we do in this paper, to identify the most pertinent characteristics of a system. In this capacity machine learning could prove to be an invaluable tool for systems engineers who are struggling to keep pace with complexity increases.

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 nearest neighbors technique, while Section 6 applies nearest neighbors to binary loop classification. Section 7 describes experiments with multi-class classification. Section 8 relates our work to previous work, and we conclude in Section 9.

2. CONTRIBUTIONS

The novel aspects of our research are summarized here (please see Section 8 for a detailed comparison of our research and related work):

- To the best of our knowledge, we are the first to 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 have tried to solve harder multi-class problems.
- We are the first to show that nearest neighbor classification is a viable method for improving compiler decisions.
- We show that the nearest neighbor approach can isolate the four most important factors for predicting when unrolling is appropriate, and more generally, for predicting the best unroll factor.
- We show that learning techniques can improve the performance of a well-respected compiler targeting a modern architecture.

We have also publicly 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

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 [5, 7] 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 [6] 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 [6] and hyperblock formation [8]. 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.

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 exit early, it will actually have to add control flow to the unrolled loop that 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 are complex tools. It is nearly impossible to know what choices to make based on simple models and assumptions. The scheduler, the register allocator, and the underlying architecture interact in mysterious ways. The only way to truly know what will work is to empirically evaluate decisions. It is the goal of this research to use empirical observations to train to a compiler to make informed decisions.

4. METHODOLOGY AND INFRASTRUCTURE

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

Feature
The language (C or Fortran).
The tripcount of the loop (-1 if unknown).
The estimated frequency of execution of the loop.
The loop nest level.
The maximum dependence height of the loop.
The average dependence height.
Number of operations.
The maximum height of memory dependencies.
The maximum height of control dependencies.
The number of parallel "computations" in loop.
The number of indirect memory references.
The number of induction variables.
Minimum num. iterations between memory loop-carried deps.
Minimum num. iterations between scalar loop-carried deps.
Number of calls.*
Number of floating point operations.*
Number of branches.*
Number of memory operations.*
Number of floating point memory operations.*
Number of distinct predicates used.
Number of hazards.*
Number of operands.*
Number of control speculation instructions.*
Number of data speculation instructions.*
Estimated cycles of critical path.*
Number of live ranges into the loop.*
Number of live ranges out of the loop.*
Number of uses in the loop.*
Number of defs in the loop.*
Number of definitions that reach the loop entrance.*
Number of definitions that reach the loop exit.*

Table 1: A subset of features used for loop classification. These characteristics are used to train the nearest neighbors classifier. The features that are marked with an asterisk are normalized by the number of operations in the loop.

4.1 Our Approach: Supervised Learning

The experiments conducted in this paper use an offline learning technique known as supervised learning. Though the learning algorithm is trained offline, the learned classifier can easily be incorporated into a compiler. 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. 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. In total, we use 38 features in these experiments, but as we show later, using many fewer features works nearly as well.

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 $(1, 2, \ldots, 8)$. The label for the loop is the unroll factor that yields the best performance.

To reiterate, for each loop — alternatively referred to as a training example — 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.

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).

4.2 Computing the Accuracy

The accuracy numbers presented in this paper are computed using a methodology known as leave-one-out cross-validation (LOOCV). 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 data set. On each iteration i, the algorithm 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.

Section 5 describes the learning technique that we use to perform the experiments in this paper. The remainder of this section discusses our infrastructure and how we collect training labels.

4.3 Compiler and Platform

We used the Open Research Compiler (ORC v2.1) [10]— 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. We disable software pipelining to focus on the loop unrolling heuristic, and we set the maximum unroll factor to eight.

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 a great deal of 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 prolog and epilog 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's finalize procedure. This procedure prints 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 was most beneficial for each loop, and thus these cycle counts form the basis of our labeled training data set.

We realize that we cannot possibly measure loop runtimes without affecting the execution in some way. However, we have made an earnest — and we think successful — attempt at hiding the overhead of loop instrumentation. Nevertheless, to further mitigate the impact of the instrumentation on the accuracy of the runtime numbers, we only use loops that are run for at least 50,000 cycles. This helps reduce the amount of noise in our training data set. 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 take the median runtime for a given unroll factor.

4.5 Benchmarks Used

We use the benchmarks listed in Table 2 to gauge the efficacy of our approach[1]. The benchmarks come from a variety of benchmark suites and span three languages (C, Fortran, and Fortran90). The Table shows the number of loops that each benchmark contributes to our training set. In some cases, only a small fraction of the loops in a benchmark are included in our training set. There are three main reasons for this: many of the loops are not unrollable[2], we only use loops that are run for a minimum of 50,000 cycles, and we only use loops where the best unroll factor is measurably better (1.1x) than the average over all unroll factors. The latter two criteria are intended to filter out noisy examples that might prevent a learning algorithm from generalizing.

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

Benchmark	Loops	Category
008.espresso	50	Logic minimization.
022.li	0	Lisp interpreter.
052.alvinn	0	Neural network training.
099.go	37	Go-playing program.
101.tomcatv	5	Vectorized mesh generation.
124.m88ksim	7	Motorola 88100 simulator.
129.compress	6	Compression.
146.wave5	51	Maxwell's equations.
164.gzip	10	Compression.
168.wupwise	11	Physics simulations.
171.swim	8	Shallow water modeling.
172.mgrid	4	Multi-grid solver.
173.applu	60	PDE solver.
175.vpr	11	Circuit placement and routing.
176.gcc	83	C compiler.
177.mesa	5	3-D graphics library.
178.galgel	246	Fluid dynamics.
179.art	20	Image recognition.
181.mcf	2	Combinatorial optimization.
183.equake	9	Seismic wave simulation.
187.facerec	21	Face recognition.
188.ammp	5	Computational chemistry.
189.lucas	12	Number theory.
197.parser	26	Word processing.
200.sixtrack	33	Particle accelerator simulation.
255.vortex	5	Object-oriented database.
256.bzip2	17	Compression.
300.twolf	43	Place and route simulator.
301.apsi	47	Meteorology simulator.
QCD	11	Quantum chromodynamics.
TRACK	20	Missile tracking.
BDNA	22	Molecular dynamics simulation.
OCEAN	44	2-D ocean simulation.
MG3D	45	Depth-migration code.
TRFD	13	Two-electron integral transformation.
linpack	7	Linear equation solver.
mmmul	1	Matrix-matrix multiply.
purdue bench	27	Synthetic parallel benchmarks.
vector	112	Test suite for vectorizing compilers.
whetstone	2	Synthetic benchmark.
Total	1138	

Table 2: Benchmarks used. This table lists the benchmarks from which loop runtimes are extracted, as well as the number of loops contributed by each benchmark. We only use loops that ORC can unroll and whose optimal unroll factor is measurably better than the average (1.1x) over all unroll factors up to eight. Note that the purdue benchmark entry is actually comprised of nine separate benchmarks.

5. NEIGHBOR CLASSIFICATION

Nearest neighbor (NN) classification is an extremely intuitive learning technique. 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 finding the nearest example in the database and using its label. 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 a nearest neighbor classifier on real loop data. Each of the points in the figure represents a loop from our suite of benchmarks. 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 ($\mathbf{x_i}$) —

[1]Please note that we have excluded some SPEC benchmarks. There are two main reasons for this: some of the benchmarks did not compile properly with ORC and our loop instrumentor (the loops have to run correctly for *all* unroll factors), and we simply ran out of time. We are still adding benchmarks to our training set, and as with most learning approaches, our generalization accuracy should improve with the size of our training set.

[2]ORC only unrolls single block inner-loops. In practice, this is not overly restrictive as a hyperblock formation phase precedes the loop unrolling phase.

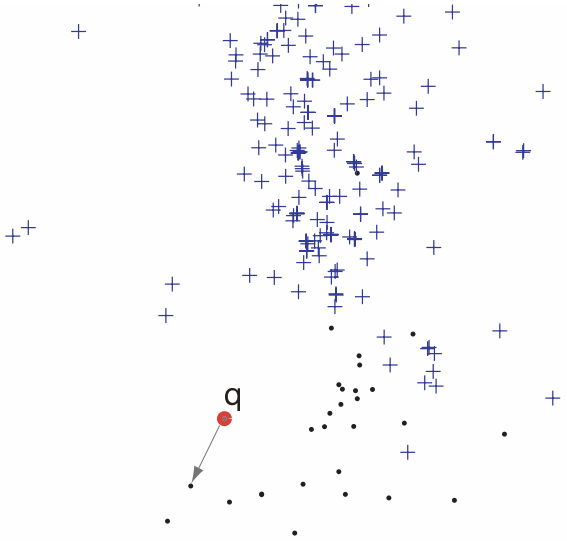

Figure 1: Nearest neighbor classification. This figure highlights the salient features of the nearest neighbors algorithm. Each of the points in the figure corresponds to a 2-D projection of normalized loop features. The blue crosses correspond to those loops that should be unrolled (according to empirical evaluation) and the black dots correspond to those that should not. By using this 'database' of loops, nearest neighbors can quickly predict an action for a new loop. Here we query the database with loop q to determine that it should not be unrolled. To improve the visualization in two dimensions, this figure only considers loops where unrolling either degrades or improves performance by over 30%.

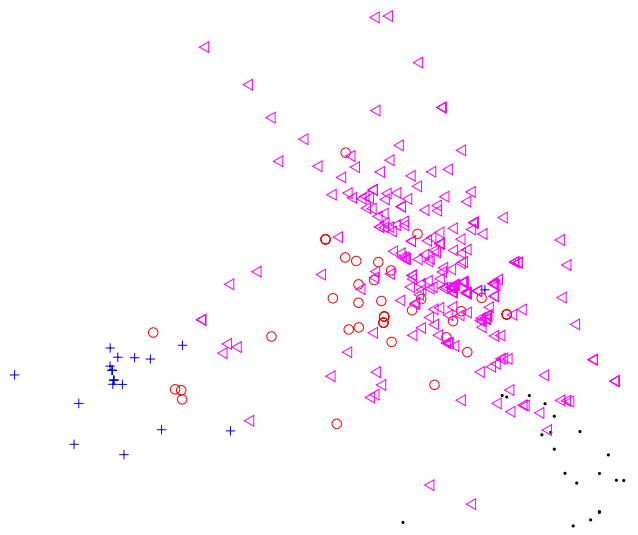

Figure 2: Nearest neighbor classification into many classes. Even when projected from a high dimensional space onto a plane, we see that loops that the compiler should treat similarly cluster together. The dots, the crosses, the circles, and the triangles represent loops with unroll factors of one, two, four, and eight respectively.

onto a plane[3].

The nearest neighbors algorithm makes predictions for a new point based on the value of the point's nearest neighbor in the database. In Figure 1, the query point q is nearest to a point that has been empirically identified as a loop that should not be unrolled. Therefore, the algorithm would predict that this loop remain rolled.

Note that NN classification is trivial to train: we simply have to populate a 'database' of examples. Though the training time of a classifier is not a tantamount 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 set sizes — which ours is — the lookup is extremely fast[4].

6. BINARY CLASSIFICATION

Before considering the case in which we attempt to determine the best unroll factor, it is instructive to first try to determine whether unrolling a given loop is beneficial.

[3]Note that the axes of the graph correspond to a linear combination of the dimensions in the original feature space.

[4]With over 1100 examples in our database, the linear-time scan takes less than 3ms. Lookup time is far outweighed by compiler fixed-point dataflow analyses.

Algorithm	Accuracy
Nearest Neighbors	0.88
Always predicting unroll	0.77
ORC's decision	0.72

Table 3: Accuracy of binary predictors for loop unrolling. This table compares the accuracy of three different predictors: nearest neighbor classification, always predicting unroll, and ORC's prediction.

Table 3 summarizes the results. Nearest neighbors predicts with 88% accuracy whether *all* unroll factors are better than not unrolling. Thus, regardless of what unroll factor ORC chooses, unrolling will be beneficial for positive examples. Note that if we were to always predict unroll, we would only be right 77% of the time for the 1138 loops in our dataset. ORC predicts correctly 72% of the time.

If instead we try to predict whether unrolling using ORC's unroll factor is beneficial, nearest neighbors correctly predicts 89% of the examples. This marginal increase in accuracy is not surprising since the vast majority of the time, not unrolling is either among the best decisions or the worst decisions (83% of the time it is either the best decision or the worst decision). Thus, whatever unroll factor ORC chooses will probably be better than not unrolling when some unroll factors are beneficial; the converse is also true.

6.1 The Best Four Features

The NN algorithm can be trained extremely quickly, as it only involves populating the database of examples. It also classifies examples quickly for small databases. We therefore experimented with exhaustively searching for the most informative four features for NN classification. In this capacity machine learning techniques can help engineers identify the

most influential aspects of their systems.

With four features, the classifier can predict with 85% accuracy when to unroll. The four best features are summarized here:

- The number of operations in the loop body: It is not surprising that this is one of the best features. Large loop bodies will not likely expose any exploitable intra-iteration parallelism and will significantly increase register pressure when unrolled.
- The maximum critical path height: Unrolling loops with long critical paths will not significantly expose ILP because such computations are sequential.
- Minimum memory to memory loop carried dependency: If a memory access is dependent on a memory access from the previous iteration, this will hinder code motion opportunities when the loop is unrolled. Thus, it may make sense to leave the loop rolled.
- The number of indirect memory references: The following example from bzip2 illustrates why this feature might be valuable:

  ```
  rfreq[bt][szptr[i]]++
  ```

  When the loop is unrolled, the indices into the `rfreq` array can be computed in parallel at the top of the loop. By loading the indices early, the loop's runtime can be drastically reduced. On the other hand, ORC and Itanium support data speculation; memory aliasing ambiguities introduced by indirect memory references may cause ORC to insert speculative memory operations that can impair performance in many cases.

7. MULTI-CLASS CLASSIFICATION

While knowing whether a loop should be unrolled is helpful, knowing the optimal unroll factor is even better. In this section we describe the operation of a multi-class classifier for loop unrolling. Thus, instead of trying to classify a loop into one of two categories, we will now use eight categories, corresponding to unroll factors one through eight. Recall that an unroll factor of one leaves the loop rolled.

As with the two-class case, 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.

NN is used for multi-class classification in the same manner as described above for the two-class case. We train the NN algorithm the same way we did for the two-class case, except now the predicted unroll factor for a novel loop will be the unroll factor for the loop to which it is nearest.

Table 4 shows the accuracy of the learning algorithm and ORC's heuristic. Using leave-one-out cross validation we find that 60% of the time the NN algorithm 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 infer from the table that a full 74% of the time, NN classification is within 6% of the optimal performance.

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

Prediction Correctness	NN's	ORC's	Cost
Optimal unroll factor	0.60	0.16	1x
Second-best unroll factor	0.14	0.21	1.06x
Third-best unroll factor	0.10	0.21	1.16x
Fourth-best unroll factor	0.05	0.13	1.26x
Fifth-best unroll factor	0.03	0.16	1.31x
Sixth-best unroll factor	0.03	0.04	1.31x
Seventh-best unroll factor	0.02	0.05	1.73x
Worst unroll factor	0.03	0.04	1.61x

Table 4: Accuracy of predictions for the nearest neighbors algorithm 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. Nearest neighbors predicts the optimal or nearly-optimal unroll factor 74% of the time. The *Cost* column shows the runtime penalty for mispredicting (as compared to the optimal factor).

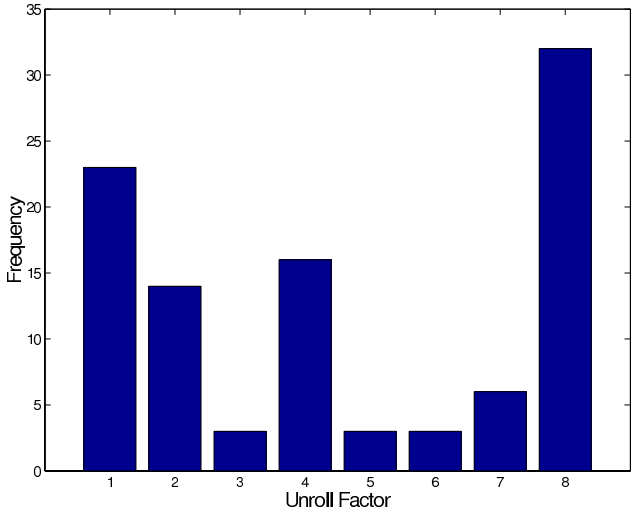

Figure 3: Histogram of optimal unroll factors. This figure shows the percentage of loops for which the given unroll factor is optimal. The histogram was constructed from 1138 loops spanning several benchmarks suites.

7.1 Realizing Speedups

In this section we see if improved unrolling classification accuracy yields program speedups. For these experiments, we compile each of the benchmarks in Table 2 using the NN algorithm to compute an unroll factor for each loop. We do **not** instrument the compiled code for the experiments in this section; we simply use the UNIX `time` command and the median of three trials to measure whole-program runtimes. Similar to LOOCV, when compiling a benchmark, we **exclude** all examples from that benchmark in the NN database. In this way we see how well the learned compiler algorithm performs on loops that it has not seen before.

Figure 4 shows the performance improvement of the NN algorithm over ORC's unrolling heuristic. The figure also shows the speedup that the compiler could obtain if an oracle were to make its unrolling decisions. NN achieves a speedup

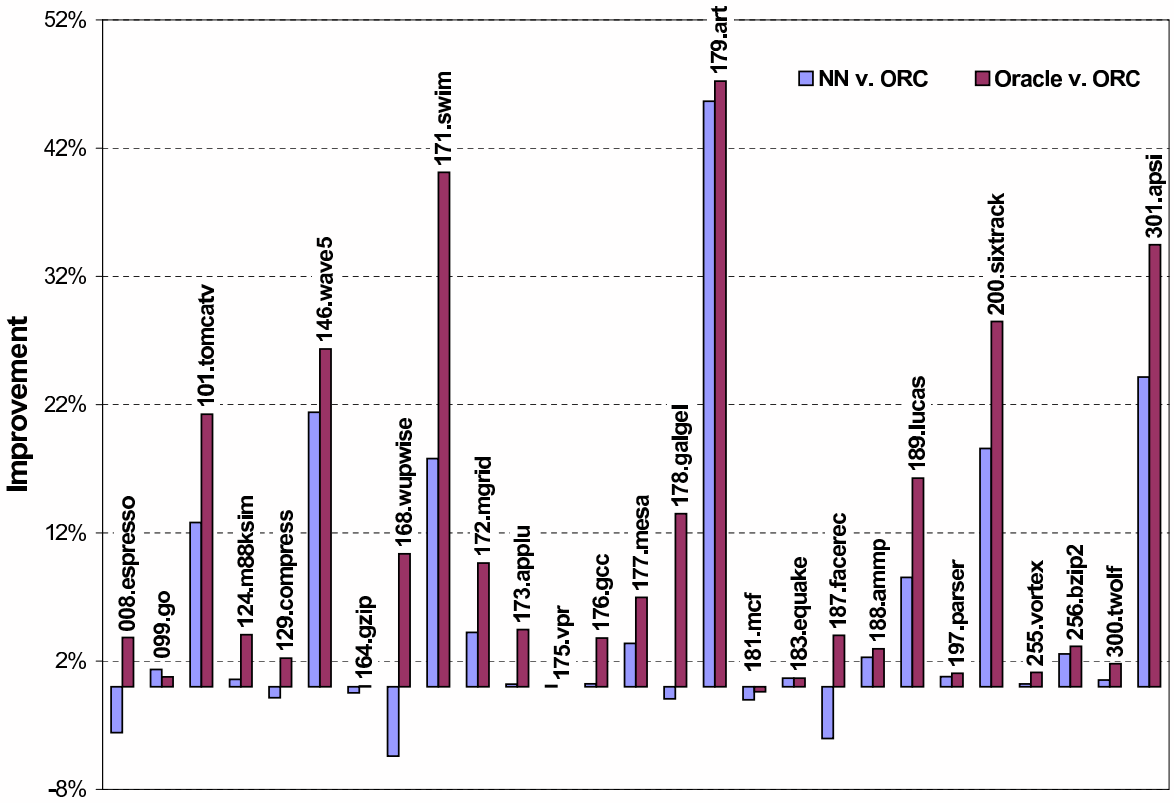

Figure 4: Realized performance on the SPEC benchmarks. We attain speedups on 20 of the 27 benchmarks in this graph, and a 6% speedup overall (5% using the geometric mean). The rightmost bar for each benchmark shows the speedup that a perfect classifier would attain.

for most of the SPECs that we include. We achieve speedups on 20 of the 27 SPEC benchmarks. Overall our technique attains a 6% overall speedup on the SPECs (5% using the geometric mean).

Figure 5 shows the performance of the remaining benchmarks in our training set. We achieve speedups on nearly all of these benchmarks. However, our predictor badly mispredicted key loops in the nas and mmmul benchmarks. For mmmul, the key loop should have been unrolled by a factor three, but instead NNs choose an unroll factor of eight. Perhaps one reason for the confusion is the fact that unroll factors of three are only optimal 3% of the time. Thus there are relatively few examples with this unroll factor in the database.

7.2 The Best Four Features

We also exhaustively found the best four features for discriminating between unroll factors. Together, the features below allow a nearest neighbor classifier to correctly classify 55% of the examples in the training set:

- The number of operations in the loop body is again one of the most important discriminating factors.
- The number of predicated operations in the loop body: This feature helps discriminate between loops with conditional control flow and simple, control-independent loops. The presence of control flow may restrict the compiler's code motion opportunities and render unrolling useless.
- The source code language: One possible explanation that this is a key feature is that Fortran code is easier to analyze than C code; the fact that C arrays can alias forces the compiler to make conservative assumptions about memory accesses, and thus some optimizations cannot be performed (or alternatively, speculative instructions must be used). Another distinct possibility is that the problems people choose to implement in Fortran are inherently more amenable to loop unrolling. Fortran has long been the language of choice for scientific computing, applications of which are typically highly parallel.
- The number of definitions that reach the loop entrance: That this feature is included is no surprise. One of the primary drawbacks of loop unrolling is the additional register pressure that it creates. This characteristic serves as a gauge for measuring the register pressure surrounding the loop body.

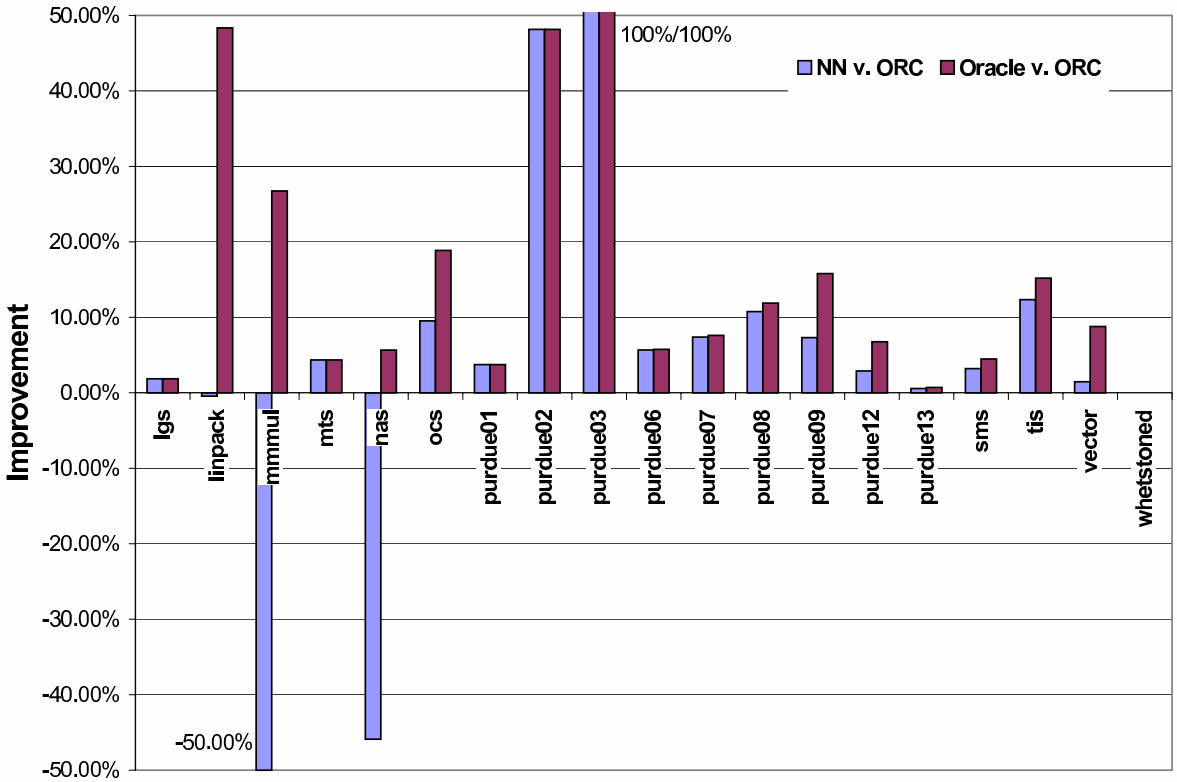

Figure 5: Realized performance on miscellaneous benchmarks. We achieve speedups on 16 of the 19 kernels and synthetic benchmarks in this figure. Overall we attain a 7% speedup on these benchmarks (4% using the geometric mean). The rightmost bar for each benchmark shows the best speedup that could possibly be attained.

8. 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 [9]. While the methodology we present in this paper is similar to theirs, our work differs in several important ways. Most importantly, our experiments employ multi-class classification to determine the optimal unroll factor. They only consider the two-class classification problem presented in Section 6, leaving the choice of unroll factor up to a compiler heuristic. Another major difference is that they unroll loops before compiling, not in the backend as we do. When trying to decide whether loop unrolling should be performed using ORC's unroll factor we arrive at roughly the same classification accuracy (our 88% compared to their 85%). These numbers may not be comparable however, because we use two completely different compiler infrastructures and learning algorithms.

Calder et al. used supervised learning techniques to fine-tune static branch prediction heuristics [1]. They employ neural networks and decision trees to search for effective static branch prediction heuristics. 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.

Stephenson et al. use genetic programming to fine-tune compiler priority functions [12]. Their unsupervised techniques seem to work well for the problems they studied, but supervised learning of the form presented in this paper is far more efficient whenever a training data set can be created. Their technique requires weeks to train, while most supervised learning algorithms require minutes or seconds. In addition, genetic programming is extremely unstable, with back-to-back runs yielding completely different results.

Cooper et al. use genetic algorithms to solve compilation phase ordering problems [4]. Their technique is well suited to the task, but cannot be extended to handle other compiler problems.

Several compiler researchers have created model-based systems to automatically compute unroll factors [11, 3, 2]. In particular Sarkar 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 this result is impressive, the models developed in [11] are much more complicated than the nearest neighbors approach we employ. While our test infrastructures are different (and probably not comparable), it is worth noting that we achieved a 9%

improvement on the SPECfp benchmarks in our training set (8% using the geometric mean).

9. CONCLUSION

Compiler developers have always had to contend with complex phenomena that are not easy modeled. For example, it has never been possible to create a useful model for all the input programs the compiler has to optimize. However until recently, most architectures— the target of compiler optimizations— were simple and analyzable. This is no longer the case. A complex compiler with multiple interdependent optimization passes exacerbates the problem. In many instances, end-to-end performance can only be evaluated empirically.

To that end, this paper experiments by using empirical evidence to teach a simple machine-learning algorithm how to make informed loop unrolling decisions. By using a large database of empirical loop observations, our technique classifies loop unrolling factors with great precision. Using leave-one-out cross-validation to find the generalization ability of the classifier (i.e., how well it performs on examples that are not in the training set), the algorithm is able to determine with 88% accuracy when loop unrolling should be performed. We show that nearest neighbors can also be used to predict the optimal unroll factor for a given loop: a full 74% of the time it predicts the optimal, or the nearly optimal solution.

We also translate these results into speedups on a real machine. Using a well-respected compiler and targeting the Itanium 2 architecture, we find that the nearest neighbors algorithm improves the performance of 36 of the 46 benchmarks in our suite. We achieve a 6% improvement on the SPEC benchmarks, and a 7% improvement on miscellaneous small benchmarks and kernels.

While we are pleased with the performance improvements, we are more excited about the complexity ramifications of our research. Apart from extracting features that we think *might* be pertinent, we thought little about designing an unrolling heuristic. Furthermore, we were able to use the learning algorithm to exhaustively reduce our large feature set to the four most important characteristics for loop unrolling.

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 at least as well as human designers. We hope that automatic heuristic tuning based on empirical evaluation will become prevalent, and that designers will intentionally expose algorithm policies to facilitate machine-learning optimization.

10. 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] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.