Compilers rely on efficiency of the intermediate passes to optimize the machine code. Owing to emergence of high performance architectures today, there are many opportunities available to the compiler for optimizing the machine code. We focus on loop unrolling, one such compiler optimization, to extract parallelism and enable better instruction scheduling. Owing to a huge search space in machine code, machine learning algorithms can account for dependencies between several optimization characteristics at once with the same runtime cost. In order to evaluate the effectiveness of machine learning techniques for optimizing compiler passes, we develop heuristics for loop unrolling using random forest algorithm. We use LLVM Compiler v3.3 to implement and evaluate our heuristics using SPEC 2006 benchmarks. We obtain a speedup of XX percent on our testing set of benchmarks over a baseline LLVM compiler. In addition, even with our limited training set, we obtain performance comparable to the loop unroll pass that is inbuilt in LLVM.

Researchers, for generations, have found efficient solutions to optimize machine code generated by a compiler. Multiple traditional optimizations such as loop inlining, loop unrolling, speculative loop invariant code motion, code block ordering, etc. enable vast opportunities to optimize machine code. Majority of these optimizations, exploit parallelism in the machine code so that the instructions can be scheduled efficiently in the modern superscalar processors. However, these optimizations are computationally intensive and can often result in degraded performance when done aggressively. As a result, it is important to regulate these optimizations by implementing intelligent heuristics to determine the sections of the code for which the optimizations can be profitable.

Emerging machine learning techniques are known to analyze a huge search space of data and implement intelligent predicting models. These techniques can evaluate multiple decision variables at once and can account for dependencies between several such variables. The goal of our project is to explore a machine learning technique as an efficient alternative to develop heuristics for compiler optimization passes. We use supervised learning technique to obtain the heuristics. Supervised learning techniques train heuristics over a labeled data consisting of a vector of heuristics and desired output value. As a case study, we develop heuristics for loop unrolling using a random forest machine learning technique. Random forest technique works on the principle of decisions trees; it generates multiple decision trees, obtains classification from each of them, and chooses the classification that gets majority of the votes. Unlike decision trees, random forests are not prone to overfitting and hence a better candidate while deploying a heuristic working on a wide variety of data set.

We train a random forest heuristic on a subset of loops in SPEC 2006 benchmarks. SPEC 2006 benchmarks are used widely by the research community to evaluate performance of both, the compiler optimizations and the underlying hardware. We divide a set of applications in SPEC 2006 benchmark in two categories training set and testing set. We generate a set of features in a loop and unroll factor that enables the best runtime of that loop over the training set to obtain a labeled data. We use this labeled data to train the heuristic and finally deploy it on testing set for evaluation.

Loop unrolling is a widely used compiler optimization pass targeting simpler loops in the program. Programs spend a significant duration of their execution time in the loops and hence, it is important to efficiently schedule instructions in a loop to exploit parallelism in them. Due to the iterative nature of the loop, control flow has to branch to the header every time the loop condition is satisfied. These branches are expensive and result in underutilization of hardware optimizations such as superscalar processors, load-store coalescing, cache locality etc. Loop unrolling technique expands the loops several times while reducing the number of iterations and consequently, diminishing the number of branches. The advantages of loop unrolling are several as listed below:
• Better instruction scheduling as the window to schedule consecutive instruction increases. Hardware can exploit higher ILP.
• Most loops load and store values to an array of memory locations. Loop unrolling can enable the coalescing of such memory operations in the hardware.
• Reduction in the number of iterations results in fewer number of branches to the loop header. The hardware can load consecutive instructions in the memory without change in the control flow.

Loop unrolling, however, results in some of the obvious side-effects that can result in degraded performance:
• A code bloat leading to higher memory pressure.
• Increased code size and limited hardware register resulting in spilling of variables on the stack. Stack is accessed slower than register resulting in degraded program performance.

• A loop can exit at multiple places; compiler has to insert complex control flow for early exits from the program.

Aggressive loop unrolling can lead to degraded performance of an application. We develop heuristics to determine the profitable loop candidates for unrolling at the compile time.

This section discusses how learning good heuristics for loop unrolling can be modeled as a supervised learning problem. It is followed by a discussion of the infrastructure that was used to perform the experiments.

Supervised learning is the machine learning task of inferring a function from *training data*. The training data consist of a set of *training examples*. Each example is a pair $< \mathbf{x_i}, y_j >$ consisting of an input feature vector, $\mathbf{x_i}$ that contains characteristics of the object under consideration and a desired output label y_j. A supervised learning algorithm analyzes the training data and produces an inferred function, which is called a classifier, such that the overall classification error on the training data is minimized. The inferred function should then be able to predict correct output value for any valid input feature vector.

For the task of loop unrolling, the feature vector captures the characteristics of the loop being unrolled. In our experiments, we used features such as trip call count, number of calls to the loop, number of operations in the loop body, etc. to characterize the loops. Feature vectors are extracted for every candidate for loop unrolling in the set of training benchmarks. For feature extraction, every loop is first assigned a unique identifier. The loops are then canonicalized such that every loop has a preheader and exactly one backedge. A subset of features that are based on profiling information and loop-carried dependencies are then extracted. Features such as trip count, that require require additional compiler optimizations like promoting memory references to register references, analyzing scalar expressions in loops etc. are extracted separately, so that they don't affect the statistics for other features. Figure 1 summarizes the features used to train the classifier in this experiment.

For the training dataset, we also extract label for each candidate loop in our train benchmark suite. The label for a loop indicates the best unrolling factor for the loop. In this study, the unroll factor (1,2,..8) yielding the best performance for the loop is used as the label for the loop. Thus, each example in the training set corresponds to a feature vector indicating loop characteristics, and a label indicating the best unroll factor derived empirically. The classifier then learns how to best map these loop characteristics, $\mathbf{x_i}$, to their corresponding unroll factor y_j. Features for loops in the test bechmark suite are extracted similarly, but since extracting labels is expensive, no labels are extracted for the test benchmark suite. Training the classifier is offline; this approach incurs no overhead at run-time.

We used LLVM (version 3.3) that provides a modern source- and target-independent optimizer, along with code generation support for many popular CPUs. The modules to extract features and labels for the loops along with the module to unroll the loop were written using LLVM. LLVM was selected as it is easy to use and integrate various analysis and transformation passes for feature extraction, label extraction and loop unrolling. The experiments in this paper were performed on 2.2 GHz Intel Xeon 64-bit 12 core server with 32KB L1 cache, 256KB L2 cache and 16MB L3 cache. For all our experiments all optimizations except for loop-simplify were disabled unless required by another analysis or transform pass.

Features
The nest level of the loop.
The number of dynamic operations in the loop body.
The number of floating point operations in the loop body.
The number of branch operations in the loop body.
The number of memory operations the the loop body.
The number of load operations the the loop body.
The number of store operations the the loop body.
The number of implicit instructions the the loop body.
The number of operands in the loop body.
The number of unique predicates in the loop body.
The number of calls to the loop
The number of definitions in the loop body
The number of uses in the loop body
The number of array element reuses in the loop body
The minimum tripcount of the loop (-1 if unknown)

Figure 1: Features used for the classification task

In order to collect labels for the loops in the train benchmark suite, the loops must be instrumented. We evaluated three approaches to unroll loops. First, if the profiling is done at program level, all different combinations of unroll factors for the loops in the program must be tried to identify the combination yielding the best performance. Owing to combinatorial considerations, this solution is computationally intractable. Second, the computational intractability in loop level profiling can be handled by analyzing the loops from innermost to outermost such that optimal unroll factor for the innermost loop is first identified. Subsequent loops are analyzed after the innermost loop is unrolled by the optimal unroll factor. However, this approach is expensive because the program being analyzed must be executed *no. of loops * no. of unroll factor* times. Also, to implement such a system is not straightforward. We instead used *loop level profiling* wherein each loop is analyzed in independently of other loops. The central idea is to instrument each loop so the execution time for each loop can be determined, and then unroll every loop in the program by an unroll factor (1,2...8). Optimal unroll factors for each loop are then determined based on their performance. Although the instrumentation at loop level is a bit intrusive, the program has to be executed only once for each unroll factor, reducing the time/cost of generating the training dataset.

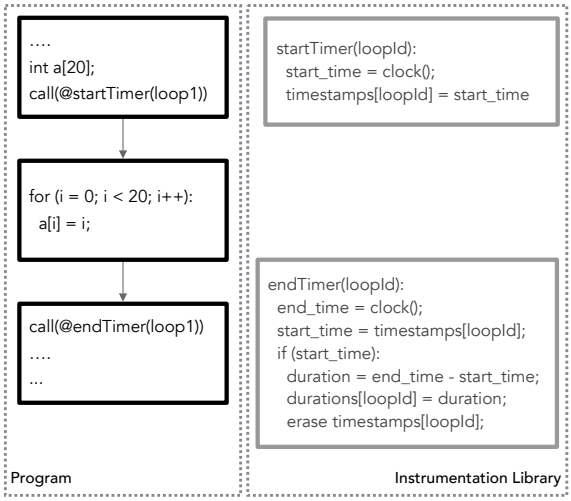

Figure 2: Instrumentation of loops

We invested much engineering effort minimizing the impact that the instrumentation code has on the execution of the program. Instead of relying on a third-party instrumentation library, we wrote our own instrumentation library consisting of two functions namely, *startTimer* and *endTimer* . Our loop instrumentor inserts an instruction that calls the start timer function at the start of the loop. An instruction to call the end timer function is inserted at each exit of the loop. In our preliminary experiments, we printed the running time of each loop at loop exit. But this instrumentation technique was abandoned due to high file I/O overhead incurred at each loop exit. This problem was aggravated for inner loops that are executed more frequently than outer loops. To reduce this overhead, 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. Figure 2 shows how the calls to the library are inserted at the start and exits of the loop.

We realize that we cannot possibly measure loop runtimes without affecting the execution in some way. However, since the instrumentation will affect loops with all unroll factors similarly, it does not affect the choice of optimal unroll factor for a loop.

To extract features for loops in the train benchmarks, we run each benchmark through a series of passes that assign a unique label to the loop, canonicalize the loop, profile the edges in the loop, extract profile-based features and extract trip count of the loop. The benchmark is then instrumented and simulation times are found for all unroll factors up to eight; an unroll factor of one corresponds to leaving the loop intact (rolled). For each loop the optimal unroll factor is determined based on the runtime for each unroll factor. Figure 3 shows the system architecture of HOLA. Once the training data is collected, a classifier is trained to learn the mapping from loop characteristics to optimal unroll factor. The trained classifier is then used to predict best unroll factor for the loops in the test benchmark suite based on their features extracted the same way as the features for loop in train benchmark suite. The predicted optimal unroll factors for each loop in the test benchmark suite are then applied to the corresponding loops. The test benchmarks are then compiled and executed to assess improvement in performance.

We develop a random forest trained heuristic for Open Source LLVM Compiler version 3.3. We implement all the custom passes to unroll and instrument loops and to collect features in LLVM. In addition, we use SPEC 2006 integer and floating point C++ applications to train and test our heuristics. Due to the limited resources available to us, we train our heuristics on the set of applications having lower number of loops. We train our heuristics on a training set including 470.lbm, 462.libquantum, 458.sjeng, 456.hmmer, 429.mcf, 401.bzip2, and 445.gobmk applications. Our testing set involves rest of the SPEC 2006 applications. We train and test our heuristics on Intel Xeon 64-bit 2.2GHz 12-core server with 32KB L1 cache and 256KB L2 cache.

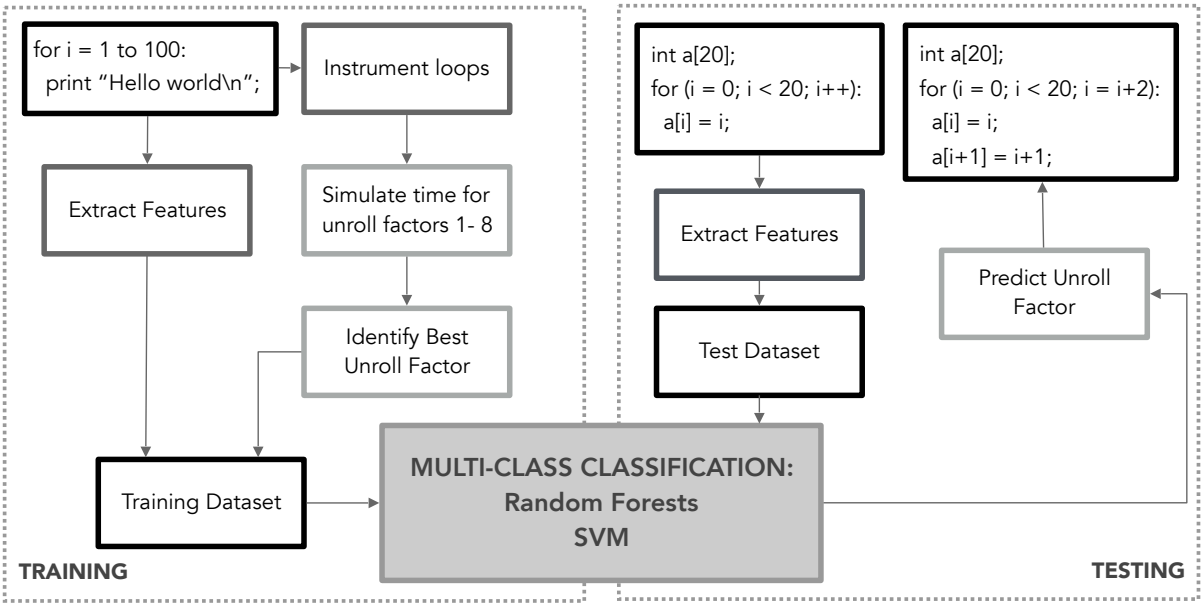

Figure 3: System Architecture.

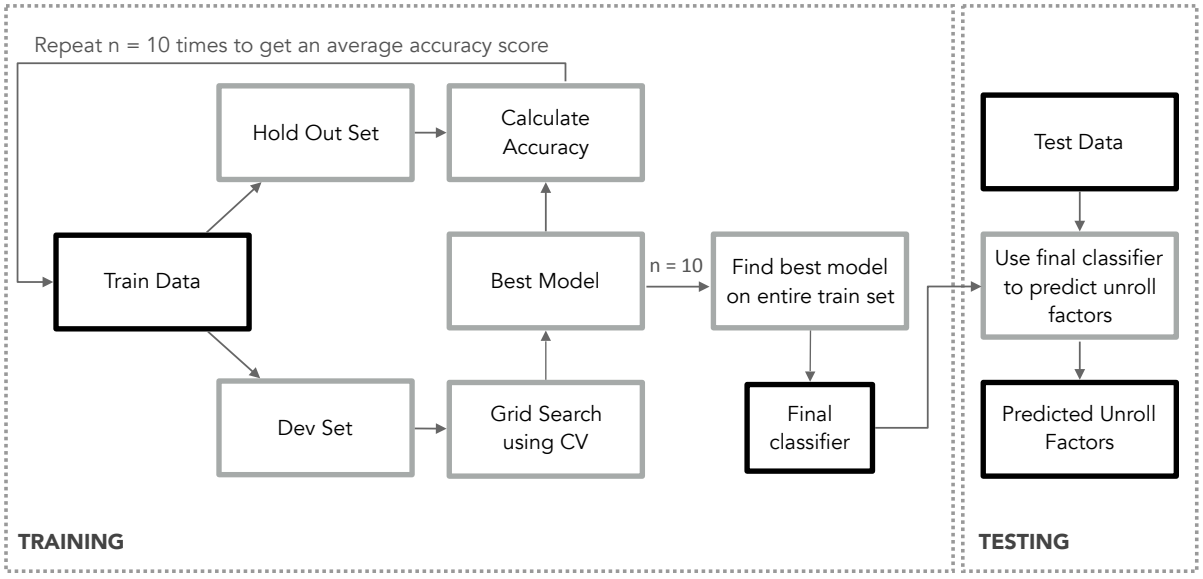

Figure 4: System Architecture.