Inducing Heuristics To Decide Whether To Schedule

John Cavazos
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003-9264, USA
cavazos@cs.umass.edu

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003-9264, USA
moss@cs.umass.edu

ABSTRACT

Instruction scheduling is a compiler optimization that can improve program speed, sometimes by 10% or more—but it can also be expensive. Furthermore, time spent optimizing is more important in a Java just-in-time (JIT) compiler than in a traditional one because a JIT compiles code at run time, adding to the running time of the program. We found that, on any given block of code, instruction scheduling often does not produce significant benefit and sometimes degrades speed. Thus, we hoped that we could focus scheduling effort on those blocks that benefit from it.

Using supervised learning we induced heuristics to predict which blocks benefit from scheduling. The induced function chooses, for each block, between list scheduling and not scheduling the block at all. Using the induced function we obtained over 90% of the improvement of scheduling every block but with less than 25% of the scheduling effort. Deciding when to optimize, and which optimization(s) to apply, is an important open problem area in compiler research. We show that supervised learning solves one of these problems well. to our training set.

1. INTRODUCTION

It is common for compiler optimizations to benefit certain programs, while having little impact (or even a negative impact) on other programs. For example, instruction scheduling is able to speed up certain programs, sometimes by 10% or more [15]. Yet on other programs, applying instruction scheduling has little impact (and in some rare cases, degrades performance). Reasons for this are that equivalent orderings happen to execute at the same speed, or because the block has only one legal order, etc.

If instruction scheduling was an inexpensive optimization to apply we would apply it to all blocks without regard to whether it benefits a particular block. However, scheduling is a costly optimization, accounting for as much as 10% of total compilation time in an optimizing JIT compiler. Because it is costly and because it is not beneficial to many blocks (or even entire programs), we want to apply it selectively.

We would prefer to apply scheduling to those blocks that: 1) account for a significant part of a program's running time, and 2) will benefit from applying scheduling to them.

Determining the first property is done through profiling and is a well-studied area of research [2, 18, 4]. On the other hand, determining the second property, *before* applying scheduling, has received relatively little attention. Also, to our knowledge this is the first application of supervised learning to determine whether to apply an optimization, in our case instruction scheduling.

We present in this paper a technique for building heuristics, which we call *filters*, that accurately predict which blocks will benefit from scheduling. This allows us to *filter* from scheduling the blocks that will *not* benefit from this optimization. Since in practice a large fraction of blocks do not benefit from instruction scheduling, and since the filter is much cheaper to apply than instruction scheduling itself, we significantly decrease the compiler's time spent scheduling instructions.

We show that *inexpensive* and *static* features can be successfully used to determine whether to schedule or not.

1.1 Instruction scheduling

Consider the sequence of machine instructions that a Java JIT compiler emits when it compiles a Java method. A number of permutations of this sequence may, when executed, produce the same result as the original—but different permutations may execute at different speeds. To improve code speed, compilers often include an *instruction scheduler*, which chooses a semantically equivalent, but one hopes faster, permutation. Permutations are *semantically equivalent* if all pairs of dependent instructions occur in the same order in both permutations. Two instructions are *dependent* if they access the same data (register, memory, etc.) and at least one writes the data, or if at least one of the instructions is a branch.[1]

We consider *list scheduling* over *basic blocks:* sequences with one entry and one exit. List scheduling is a traditional and widely used instruction scheduling method [10]. We used the *critical path scheduling* (CPS) model [15] in our list scheduler implementation. List scheduling works by starting with an empty schedule and then repeatedly appending *ready* instructions to it. An instruction I is ready if every instruction upon which I depends is already in the schedule. If there is more than one ready instruction, CPS chooses the one that can start soonest. If there is a tie, CPS chooses the instruction that has the longest *(weighted) critical path* to the end of the block, i.e., the path of dependent instructions that takes the longest to execute. While we offer this description of our scheduler for those who are interested, we note that our filtering technique applies to any competent scheduler: in essence we are discriminating between those blocks that a scheduler can improve significantly

[1]With additional analysis (and insertion of compensation code when necessary), sometimes schedulers can safely move instructions across a branch [8, 7]. We do not pursue that further here.

and those that it cannot, and this has more to do with the block than with details of the scheduler, provided the scheduler is generally competent at making some improvement if it is possible.

2. PROBLEM AND APPROACH

We want to construct a filter that with high effectiveness predicts whether scheduling a block will benefit an application's running time. The filter should be significantly cheaper to apply than instruction scheduling; thus we restrict ourselves to using properties (*features*) of a block that are cheap to compute. To our knowledge, this is the first time anyone has used *static* features to apply instruction scheduling selectively, so we had no "good" hand-coded heuristics to start from. We opted not to construct filters by hand, but instead to try to induce them automatically using *supervised learning* techniques.

Developing and fine-tuning filters manually requires experimenting with different features (i.e, combinations of features of the block). Fine-tuning heuristics to achieve suitable performance is therefore a tedious and time-consuming process. Machine learning, if it works, is thus a desirable alternative to manual tuning.

Our approach uses a technique called *rule induction* to induce a filter that is based on the features of the block. Other researchers have applied unsupervised learning, specifically genetic programming, to the problem of deriving compiler heuristics automatically, and argued for its superiority [17]. In contrast, we found that supervised learning is not only applicable to this problem, but preferable because (1) it is faster,[2] simpler, easier to use with rule induction (giving more understandable heuristic functions), and easier to make work (than unsupervised learning).

We emphasize that while scheduling involves a heuristic to choose which instruction to schedule next, the learning of which we have considered elsewhere [14], our goal here is to learn to *choose between scheduling and not scheduling*, not to induce the heuristic used by the scheduler. In other words, the research here involves learning *whether* to schedule, while that previous research involved learning *how* to schedule. We now consider the features, the methodology for developing the training instances, and the learning algorithm in more detail.

2.1 Features

What properties of a block might predict its scheduling improvement? This aspect of applying machine learning is more an art than a step-by-step procedure. It is the part of machine learning that least yields to having a procedure that is guaranteed to produce a good result. It is not uncommon to need to iterate a number of times in developing features for a given problem. However, we believe it is much easier to develop features of a problem than to come up with interesting combinations of features (i.e., heuristics) that are successful at solving a problem.

One can imagine that certain properties of the block's dependence graph (DAG) might predict scheduling benefit. However, building the DAG is an expensive phase that can sometimes dominate the overall running time of the scheduling algorithm [15]. Since we require cheap-to-compute features, we specifically choose not to use properties of the DAG. Instead, we try the simplest kind of cheap-to-compute features that we thought might be relevant. Computing these features requires a single pass over the instructions in the block.

We grouped the different kinds of instructions into 12 possibly overlapping categories, where instructions in each category have similar scheduling properties. Rather than examining the *structure* of the block, we consider just the *fraction of instructions of each category* that occur in the block (e.g., 30% loads, 22% floating point, 5% yield points, etc.). We also supply the block size (number of instructions in the block). See Table 1 for a complete list of the features. "Hazards" are possible but unusual branches, which disallow reordering. These features are as cheap to compute as we can imagine while offering some useful information. It turns out that they work well. We present all of the features (except block size) as ratios to the size of the block (i.e., fraction of instructions falling into a category, rather than the number of such instructions). This allows the learning algorithm to generalize over many different block sizes.

[2] Our technique induces heuristics in seconds on one desktop computer. Stephenson et al. report taking days to induce heuristics on a cluster of 15 to 20 machines.

Feature	Type	Meaning
bbLen	BB size	Number of Instructions in the block
Category		Fraction of instructions that ...
Branch	Op kind	are Branches
Call	Op kind	are Calls
Load	Op kind	are Loads
Store	Op kind	are Stores
Return	Op kind	are Returns
Integer	FU use	use an Integer functional unit
Float	FU use	use a Floating point functional unit
System	FU use	use a System functional unit
PEI	Hazard	are Potentially Excepting
GC	Hazard	are Garbage Collection points
TS	Hazard	are Thread Switch points
Yield	Hazard	are Yield points

Table 1: Features of a basic block.

We could have potentially refined our set of features by including more of different kinds, but what we have works well. We note that coming up with features for other optimizations might be easy or hard, depending on the optimization. In this case, we were "lucky" in that a little domain knowledge allowed us easily to develop a set of features that produced highly-predictive heuristics. Through our experience with identifying features and using machine learning, we noticed some useful and (possibly obvious) general principles.

1. Experiment with the simplest features first. In this case, it would have been moot to develop additional features.
2. Normalize features and simplify them. Prefer categorical or boolean values over integral or continuous ones. Binning of continuous values can also help the learning task: it simplifies and also tends to enhance readability of the induced heuristic.
3. Examine any relevant hand-coded heuristics. This not only helps in identifying important features to use, but allows us to see the underlying structure of successful heuristics, which will give clues as to how the features should be represented and used.
4. Apply the simplest learning algorithm possible to start with. Obviously, a procedure that is easier to get working (i.e., require less tweaking) is preferable.

2.2 Learning Methodology

Determining what features to use is an important (and possibly the most difficult) step in applying supervised learning to a problem. Once we determine the features, we can generate positive and negative examples for training the supervised learning component. These *training instances* consist of a vector of feature values, plus a boolean classification label, i.e., *LS* (Schedule) or *NS* (Don't Schedule), depending on whether or not the block benefits from scheduling.

Our procedure is as follows. As the Java system compiles each Java method, it divides the method into blocks, which it presents to the instruction scheduler. We instrument the scheduler to print into a trace file raw data for forming instances, consisting of the features of the block and an estimate of the block's cost (number of cycles) without scheduling, and an estimate of the block's cost with list scheduling applied.

We obtain these estimates of block cost from a simplified machine simulator. The simulator makes a number of simplifying assumptions, partly for speed but also because it is hard to determine what the state of the machine will be at run time at the moment the machine begins to execute a particular block (and that state may be different for different executions of the same block). The exact cycle estimate is not crucial; rather, the estimate needs only to give a good sense of the *difference* in timing between two versions of the same block. Note that on modern processors timing of small code fragments is not only difficult, it is not clear that it is meaningful, because there may be tens, even hundreds, of instructions in flight, with execution overlapped. It is not clear how one would further "validate" our simplified simulator: that our overall procedure produces good results is itself evidence of adequacy of the estimator.[3]

We label an instance with LS if the estimated time after list scheduling is more than t% less than before scheduling. We label an instance with NS if scheduling is not better (at all). We do not produce a training instance if the benefit lies between 0 and t%. We call t the *threshold value*. We first consider the case $t = 0$ and discuss positive threshold values later. Typically we obtain thousands of instances for each program (one for each block in the program). Table 5 shows training set sizes for different threshold values for SPECjvm98.

We apply a learning algorithm to the training instances; the output of the learning algorithm is a heuristic function: given the features of the block, it indicates whether or not we should schedule the block. It is important to note that the procedure above (including learning) occurs entirely *offline*.

The final step involves installing the heuristic function in the compiler and applying it *online*. Each block from each method that is compiled by the optimizing compiler is considered as a possible candidate for scheduling. We compute features for the block. The cost of computing the features is included in all of our actual timings. It is small relative to scheduling and to the rest of the cost of compiling a method. If the heuristic function says we should schedule a block, we do so.

2.3 Learning Algorithm

An important rule in applying machine learning successfully is to try the simplest learning methodology that might solve the problem. We chose the supervised learning technique called *rule set induction*, which has many advantages over other learning methodologies. The specific tool we use is Ripper [5].

It is easy and fast to tune Ripper's parameters (typically an important part in obtaining the best result). Ripper generates sets of if-then rules that are more expressive, more compact, and more human readable (hence good for compiler writers) than the output of other learning techniques, such as neural networks and decision tree induction algorithms. We analyze one of the induced if-then rule sets (a filter) in Section 4.6.

[3]The estimator is also used by the list scheduler as it makes decisions, but that usage is irrelevant to our learning procedure.

Benchmarks	Description
`compress`	Java version of 129.compress from the SPEC CPU95 suite
`jess`	Puzzle solving expert system shell based on NASA's CLIPS system
`db`	Builds an in-memory database and performs various operations on it
`javac`	A Java source code to bytecode compiler from JDK 1.0.2
`mpegaudio`	Decodes an MPEG-3 audio file
`raytrace`	A raytracer that works on a scene depicting a dinosaur
`jack`	A Java parser generator with lexical analysis

Table 2: Characteristics of the SPECjvm98 benchmarks.

2.4 Benchmarks

We examine 7 programs drawn from the SPECjvm98 suite [16] in our first set of experiments. We detail our chosen benchmarks in Table 2. We ran these benchmarks with the largest data set size (called 100).

3. EVALUATION METHODOLOGY

As is customary in evaluating a machine learning technique, our learning methodology was leave-one-out cross-validation: given a set of n benchmark programs, in training for benchmark i we train (develop a heuristic) using the training set (the set of instances from the $n-1$ *other* benchmarks), and we apply the heuristic to the test set (the set of instances from benchmark i). This makes sense in our case for two reasons:

1. We envision developing and installing of the heuristic "at the factory", and it will then be applied to code it has not "seen" before.[4]
2. While the end goal is to develop a single heuristic, it is important that we test the overall procedure by developing heuristics many times and seeing how well they work. The leave-one-out cross-validation procedure is a commonly used way to do this. Another way is repeatedly to choose about half the programs and use their data for training and the other half for testing. However, we want our heuristics to be developed over a wide enough range of benchmarks that we are likely to see all the "interesting" behaviors, so leave-one-out may be more realistic in that sense.

To evaluate a filter on a benchmark, we consider three kinds of results: *classification accuracy*, *scheduler running time*, and *application running time*.

[4]One could provide tools to end users so that they could develop their own training sets and retrain. This would be valuable only if they are likely to come up with a significantly different function, which would have significantly different performance. If we train over a large enough set "at the factory", then we presumably "cover" all the interesting behaviors of our compiler and a variety of blocks that present a full range of scheduling issues. Thus, it is not clear that user retraining would have much value. This is something we could explore using additional experimental data, such as training on an individual program and testing on that same program, which gives a kind of upper bound on how much improvement you could get by retraining.

Classification accuracy refers to the accuracy of the induced filter on correctly classifying a set of labeled instances. Classification accuracy tells us whether a filter heuristic has the potential of being useful, however, the real measure of success lies in whether applying the filter can successfully reduce scheduling time while not adversely affecting the benefit of scheduling to application running time. In a few cases, using a filter improved application running time over always applying the scheduler (this occurs when filters inhibit scheduling that actually degrades performance).

Scheduler running time refers to the impact on compile time, comparing against not scheduling at all, and against scheduling every block. Since timings of our proposed system include the cost of computing features and applying the heuristic function, this (at least indirectly) substantiates our claim that the cost of applying the heuristic at run time is low.

Application running time (i.e., without compile time), refers to measuring the change in execution time of the scheduled code, comparing against not scheduling and against scheduling every block. This validates not only the heuristic function but also our instance labeling procedure, and by implication the block timing simulator we used to develop the labels. What we can verify with this is that we have not undermined the scheduler; the scheduler can still improve some programs a lot while having little impact on others.

The goal is to achieve application running time close to the best of the fixed strategies, and compilation time substantially less than scheduling every block.

3.1 Experimental infrastructure

We implemented our instruction schedulers in Jikes RVM, a Java virtual machine with JIT compilers, provided by IBM Research [1]. Jikes RVM does not have an interpreter: all bytecodes are compiled into native code before execution. The system has two bytecode compilers, a *baseline* compiler that essentially macro-expands each bytecode into machine code, and an *optimizing* compiler.

We optimized all methods at the highest optimization setting, and with aggressive settings for inlining. We used the build configuration called OptOpt with more aggressive inlining, which increases scheduling benefit.[5]

As mentioned previously, we apply our technique to local (basic block) scheduling, not global scheduling. We have investigated superblock scheduling in our compiler setting, and with it one can get slight (1-2%) additional improvement over local scheduling. However, superblock formation requires detailed profiling information and we did not want to require that. Also, it is in a way beside the point: we are *not* trying to build a better scheduler, but trying to decide whether to apply whatever scheduler we have. We could apply our same procedure to the superblock case, and it might provide additional evidence that we can induce heuristics that greatly reduce scheduling effort while preserving most of the benefit.

Also, we did not apply our filters to a compilation approach that identifies and optimizes only frequently executed (or *hot*) methods. Applying filters to this approach would still save a lot of scheduling time (we have no reason to believe that being "hot" and benefiting from scheduling are correlated), but the savings will be smaller as a fraction of application running time (because compile time will be smaller overall). We further note that this approach requires some kind of *profiling* data, though Jikes RVM does develop execution frequency estimates online for methods and blocks, either using counters or by sampling the program counter value with timer interrupts.[6]

Our specific target architecture is the PowerPC. We ran our experiments on an Apple Macintosh system with two 533 MHz G4 processors, model 7410. This is an aggressive superscalar architecture and represents the current state of the art in processor implementations.[7] For instruction scheduling, the 7410 *implementation* of the PowerPC is interestingly complex, having two dissimilar integer functional units and one each of the following functional units: floating point, branch, load/store, and system (handles special system instructions). It can issue one branch and two non-branch instructions per cycle, if a complicated set of conditions holds. Instructions take from one to many tens of cycles to execute.

What value does static instruction scheduling have in the face of *out-of-order* execution, etc.? We have done some investigation of older processors, which have less "dynamic" scheduling (reordering of execution in the hardware), and static scheduling does give bigger percent improvements on such architectures. However, we still see useful improvements for some programs on more recent machines. In a sense this supports our methodology: if static scheduling helps a lot sometimes, but only in a minority of cases, then it is more interesting to have a good way to choose when to apply it.

All measurements are elapsed (wall clock) times. The system infrastructure also measures elapsed time spent in the compiler, broken down by phase and individual optimization. These measurements use the bus clock rate time counter and thus give sub-microsecond accuracy; this clock register is also cheap to read, so there is little overhead in collecting the information.[8] The time to apply the filter was included in the cost we attribute to scheduling.

4. EXPERIMENTAL RESULTS

We aimed to answer the following questions: How *efficient* is scheduling using filter heuristics as compared to scheduling all blocks? How *effective* are the filter heuristics in obtaining best application performance? We ask these questions first on the SPECjvm98 standard benchmark suite and next on a suite that includes only benchmarks for which list scheduling made an impact of more than 2% on their running time.

We address the first question by comparing the time spent scheduling. We answer the second by comparing the running time of the application, with compilation time removed. To accomplish the latter, we requested that the Java benchmark iterate 6 times. The first iteration will cause the program to be loaded, compiled, and scheduled according to the appropriate scheduling protocol. The remaining 5 iterations should involve no compilation; we use the *median* of the 5 runs as our measure of application performance.

4.1 Classification Accuracy

Before presenting efficiency and effectiveness results, we offer statistics on the accuracy of the induced classifiers (for threshold values t from 0 to 50). For each benchmark, we built a filter with leave-one-out cross-validation using the set of benchmarks from which the particular benchmark in question came. The filter chooses between list scheduling and no scheduling.

[5] We set the maximum callee size to 30 bytecode instructions, the maximum inlining depth to 6, and the upper bound on the relative expansion of the caller due to inlining to be a factor of 7.

[6] If there is room in a final version of the paper for this, we can add a section applying filters to superblocks and an "adaptive" JIT compiler.

[7] The G5 processor is only just beginning to be available as of this writing, and was not available when we performed most of the research. In any case, it is at least as complex as the 7410.

[8] Applying the filtering function (heuristic) is clearly cheap, but if the reviewers feel it to be important to do so, we can break that cost out and report it.

Table 3 shows the classification errors rates of rules induced by Ripper on SPECjvm98 benchmark program test sets generated during the cross-validation tests. We also include the geometric mean of these error rates. These impress us as good error rates, and they are also fairly consistent across the benchmarks.

4.2 Simulated Execution Times

Before looking at execution times on an actual machine, we consider the quality of the induced filters (compared with always scheduling and never scheduling) in terms of the simulated running time of each benchmark. We used the block simulator to predict (and therefore label) whether a block will benefit from scheduling or not. Thus, we hoped that our filters would perform well on a metric based on time reported by the block simulator. Comparing our filters with simulated execution time helps us validate the learning methodology, and to separate validation of the learning methodology from validation of the block simulator's model of the actual machine.

We calculate the weighted simulated running time of each block by multiplying the block's simulated time by the number of times that block is executed (as reported by profiling information). We obtain the simulated running time of the application by summing the weighted simulated running time of each block. More precisely, the performance measure for program P is:

$$\mathrm{SIM}_{\pi}(P) = \sum_{b \in P} (\text{\# Executions of } b) \cdot (\text{cycles for } b \text{ under scheduler } \pi)$$

where b is a basic block and π is either using a filter, always scheduling, or never scheduling. Table 4 shows predicted execution times as a ratio to predicted time of unscheduled code. We see that the model predicts improvements at all thresholds. These improvements do not correspond exactly to our measured improvements, which is not surprising given the simplicity of the basic block time estimator. What the numbers confirm is that the induced heuristic indeed improves the metric on which we based its training instances. Thus, machine learning "worked". Whether we get improvement on the real machine is concerned with how predictive the basic block simulator is of reality, at least in relative terms.

4.3 Efficiency and Effectiveness

We now consider the quality of each induced filter for threshold $t = 0$, and then present results for the rest of the threshold values. Figure 1(a) shows the scheduling time of the L/N filters (chooses to schedule or not) relative to LS (always perform list scheduling). NS (no scheduling) is 0 since it does no scheduling work. We find that on average (geometric mean) L/N takes 38% of the time of LS (i.e., is 2.5 times faster). These numbers are also fairly consistent across the benchmarks.

Figure 1(b) shows the impact of L/N filters and LS on application running time, presented relative to NS (a value smaller than 1 is an improvement, greater than 1 a slow down). Here there is more variation across the benchmarks, with LS doing the best at .977 and L/N filters doing well at .979. Of the benefit LS obtains (2.3%), L/N obtains 93% of it. Given the substantially lower cost of L/N to run, it is preferable to running LS all the time. The results are fairly consistent across the benchmarks, though some benchmarks improve more than others.

Note that our features (and filters) do not take into account the importance of the blocks and therefore do not require profile information. Scheduling only important blocks based on profiling can do no better at improving application running time than just always scheduling (unless scheduling degrades performance). Thus, even if we used profile information to schedule only the important blocks, we could still improve application running time over L/N only by a small amount.

4.4 Filtering the Instances

While the $t = 0$ result is not bad, we suspected that we could improve the classification error rates by increasing t. (Of course for a value large enough, only the NS category would be left and the error rate would be 0%!) More significantly, we suspected that by eliminating instances where the schedulers behaved similarly, and giving the learning algorithm only those points where the choice makes a significant difference, we might improve scheduler effectiveness. We were less certain of the impact on efficiency, but thought it might increase because the training sets would have fewer LS instances, but as many NS instances. We reasoned that this would tend to induce functions that would prefer scheduling blocks less often. These speculations were borne out, as can be seen in Figures 2(a) and 2(b).

Again, we performed the experiment for the L/N protocol, varying t from 0 to 50 in increments of 5. Note that $t = 0$ is the same L/N from the previous graphs. Considering first the efficiency effects, the geometric mean shows a steady improvement as t goes from 0 to 50, from 39% to 6% of the cost of LS. This is somewhat consistent across the benchmarks, but there is definite variation. We were able to cut the scheduling effort in half, but what happened to the effectiveness? First it degraded, but at $t = 20$ it improved, offering 93% of the benefit of LS. Thereafter, it generally degrades. While the results seem sensitive to the exact value of t, the value 20 improves over straight L/N ($t = 0$). At this value, scheduling is 4.3 times faster than LS.

How does thresholding affect the size of the training sets? And how does it affect the classification by the induced heuristics? We include two tables that offer some simple statistics that show what happens. Table 5 indicate how many instances (across all the benchmarks) have label LS at the given t value. That number is constant for NS (at 37280, so we only show statistics for instances with the LS label), but drops off steadily for LS as t increases.

Table 6 show how many instances *at run time* were classified with that label by the induced heuristic. We develop the Use numbers separately for each benchmark's heuristic (using leave-one-out cross validation), applied to that benchmark's instances; the table gives the sum across the benchmarks. The sum is the same for all t values (45453), but the number of NS instances increases, and the number of LS instances steadily decreases, as t increases. This clearly explains the efficiency results. As the threshold increases, the induced rules predict more blocks not to benefit from scheduling. This result further shows that effectiveness depends on the scheduling of a rather small minority of the methods, 7.4% of them for $t = 20$. This is not surprising: in compiler optimization it is often true that an optimization has little effect on many if not most programs, but is crucial to improving a certain minority of them.

Since our noise reduction technique worked well in this case, we encourage others to explore its effectiveness in other settings. We suspect it may be helpful whenever class labels are chosen on a "best" or "better than" basis that compares a "predicted" metric (such as simulated cycle count of a block) under different treatments (scheduling and not scheduling).

4.5 Filtering Applied to Other Benchmarks

While the above result is not bad, we suspected that we would have better results focusing only on benchmarks where scheduling is beneficial. We gathered a suite of programs that benefit from scheduling through an exploration of freely available Java programs on the Internet. Table 7 offers more details about these

Threshold Values	Benchmark program							Geometric mean
	compress	jess	raytrace	db	javac	mpegaudio	jack	
0%	6.66	7.68	10.96	6.33	8.34	7.36	7.63	7.86
5%	6.52	8.33	9.02	5.87	8.92	6.94	7.03	7.53
10%	5.81	7.51	7.05	5.39	8.57	5.91	6.07	6.62
15%	5.74	6.65	6.48	5.45	6.86	5.75	5.36	6.04
20%	1.52	2.04	2.79	1.14	4.06	2.30	1.69	2.22
25%	0.93	1.16	2.47	0.65	3.08	1.75	1.34	1.63
30%	0.76	0.96	1.78	0.33	2.20	1.22	0.92	1.17
35%	0.34	0.49	1.27	0.33	1.46	1.47	1.09	0.92
40%	0.53	0.30	0.43	0.13	0.89	0.55	0.20	0.43
45%	0.15	0.03	0.23	0.10	0.22	0.22	0.03	0.14
50%	0.00	0.03	0.18	0.00	0.12	0.06	0.00	0.06

Table 3: Classification error rates (percent misclassified) for different threshold values.

Threshold Values	Benchmark program							Geometric mean
	compress	jess	raytrace	db	javac	mpegaudio	jack	
0%	84.66	92.53	93.56	88.53	96.63	90.53	97.20	91.85
5%	83.70	92.09	92.81	88.53	96.08	89.26	97.16	91.27
10%	83.92	95.76	84.60	88.53	96.92	86.84	97.35	90.39
15%	83.79	92.64	86.38	88.54	97.55	89.16	97.54	90.67
20%	87.33	95.51	89.38	92.61	97.07	87.00	97.62	92.26
25%	84.18	97.09	92.31	90.24	97.79	92.10	98.39	93.04
30%	85.45	97.69	89.68	90.34	97.77	92.26	98.97	93.04
35%	98.16	96.42	99.48	92.74	97.94	99.75	99.59	97.70
40%	92.79	98.46	99.78	99.94	98.06	97.16	99.43	97.92
45%	99.55	99.98	99.96	100.00	98.85	89.99	99.93	98.26
50%	100.00	100.00	99.95	100.00	100.00	97.64	99.89	99.64

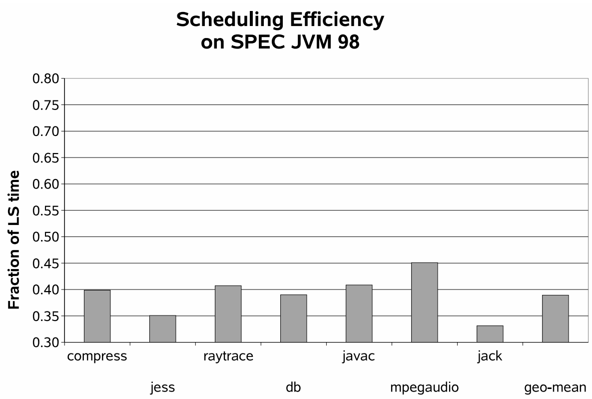
Table 4: Predicted execution times for different threshold values.

	Value of t										
	0	5	10	15	20	25	30	35	40	45	50
LS	8173	7976	7098	4930	2438	1443	912	565	316	192	49

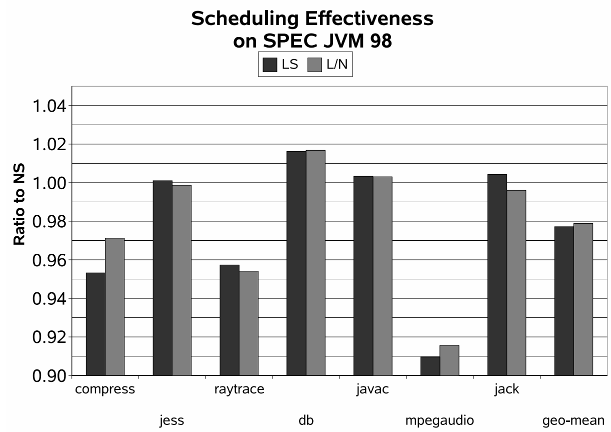
Table 5: Effect of t on training set size of SPECjvm98. NS is constant at 37280.

	Value of t										
	0	5	10	15	20	25	30	35	40	45	50
NS	39389	39256	40250	41065	42046	42557	43154	44061	44851	45142	45293
LS	6064	6197	5203	4388	3407	2896	2299	1392	602	311	160

Table 6: Effect of t on run time classification of blocks for SPECjvm98.

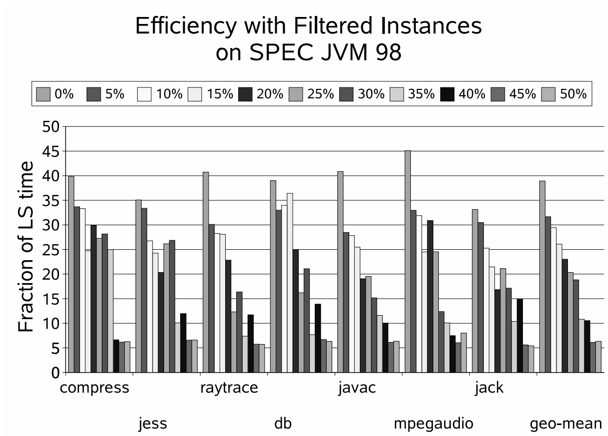


(a) Scheduling Time Using No Thresholds

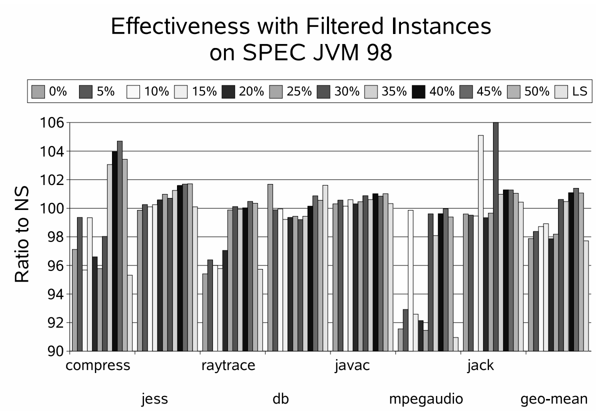


(b) Application Running Time Using No Thresholds

Figure 1: Efficiency and Effectiveness Using Filters



(a) Scheduling Time Using Thresholds



(b) Application Running Time Using Thresholds

Figure 2: Efficiency and Effectiveness Using Filter Thresholds

benchmarks. Note that this suite of benchmarks consists solely of numerically intensive (floating-point) computations. For this architecture, instruction scheduling is an important optimization for removing stalls caused by floating point instructions having long latencies.

Our reasoning for focusing on the following set of benchmarks is as follows:

If a program gains little benefit from scheduling at all, our filtering technique can reduce the compile time, but will have no substantial impact on the application running time. We could do a poor job, or a good job, of choosing which blocks to schedule, and it won't matter because the scheduler just is not having much effect.

On the other hand, if you consider a program that gets a *lot* of benefit from scheduling, then we want to make sure that we do not seriously undermine that benefit. Focusing on benchmarks that gain scheduling benefit allows us to determine this. Suppose, for the sake of argument, we included a large number of programs with little (but barely measurable) scheduling benefit. And further suppose that we show that filtering preserves that benefit. We claim that is not at all as interesting or useful as showing that we preserve the benefit gained by programs that do benefit *a lot* from scheduling. By focusing on this set of benchmarks we are trying to be *more* critical, not *less*, of our technique.

We expected that filtering would achieve most of the benefit of scheduling all blocks, while being much more efficient. This expectation was borne out, as can be seen in Figures 3(a) and 3(b).

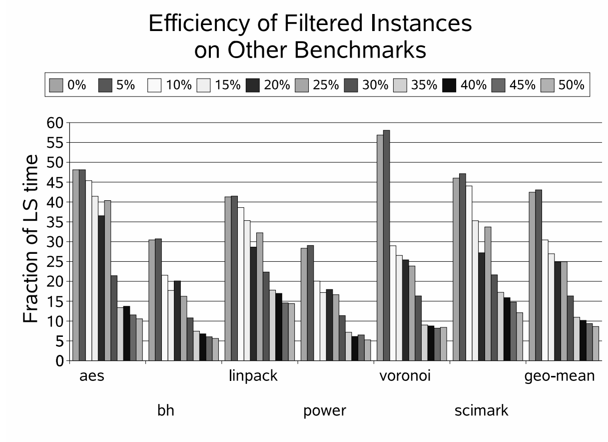
4.6 A Sample Induced (Learned) Filter

Some learning schemes produce expressions that are difficult for humans to understand, especially those based on numeric weights and thresholds such as neural networks and genetic algorithms. Rule sets are easier to comprehend and are often compact. It is also relatively easy to generate code from a rule set that will evaluate the learned filter in a scheduler.

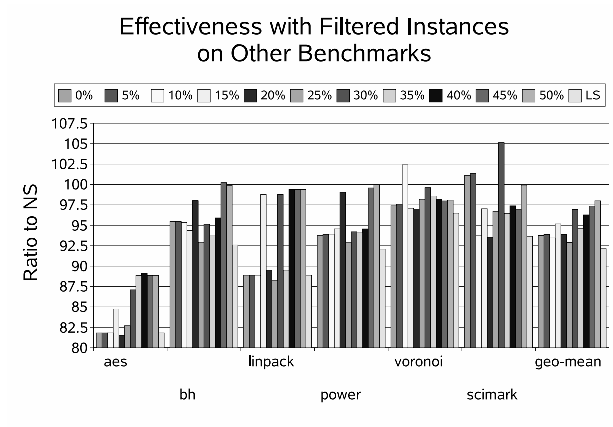
Figure 4 shows a rule set induced by training using examples drawn from 6 of 7 SPECjvm98 benchmark programs. If the right hand side condition of any rule (except the last) is met, then we will apply the scheduler on the block; otherwise the learned filter predicts that scheduling will not benefit the block.

The numbers in the first two columns give the number of correct and incorrect training examples matching the condition of the rule.

In this case we see that block size and several classes of instruc-



(a) Scheduling Time Using Thresholds



(b) Application Running Time Using Thresholds

Figure 3: Efficiency and Effectiveness Using Filters On Other Benchmarks

(924/ 12) list ← bbLen>=7	∧ calls<=0.0857	∧ loads<=0.3793 ∧ peis<=0.1493	∧ integers<=0.6087				
(661/ 8) list ← bbLen>=7	∧ systems<=0.0889	∧ stores<=0.05	∧ loads>=0.1538	∧ gcpoints<=0.0833	∧ loads<=0.5556	∧ loads>=0.3636	
(452/ 23) list ← bbLen>=7	∧ calls<=0.1034	∧ stores>=0.1778	∧ loads>=0.375				
(218/ 14) list ← bbLen>=7	∧ systems<=0.0606	∧ integers<=0.4167	∧ peis<=0.2361	∧ branches<=0.1	∧ stores<=0.0435		
(272/ 19) list ← bbLen>=7	∧ systems<=0.0741	∧ branches<=0.1111	∧ loads<=0.3667	∧ integers<=0.3667	∧ peis<=0.1667	∧ floats<=0	
(518/ 41) list ← bbLen>=7	∧ systems<=0.0606	∧ gcpoints>=0.0714	∧ integers>=0.4091				
(269/ 52) list ← bbLen>=7	∧ calls<=0.119	∧ stores<=0.0667	∧ loads>=0.2222	∧ integers<=0.2857	∧ loads<=0.625		
(74/ 3) list ← bbLen>=5	∧ stores<=0.1613	∧ loads>=0.3	∧ integers>=0.3438				
(166/ 5) list ← bbLen>=7	∧ calls<=0.119	∧ branches<=0.0476	∧ peis<=0.1765	∧ stores>=0.1237	∧ peis>=0.093		
(75/ 13) list ← bbLen>=5	∧ stores<=0.12	∧ loads>=0.2083	∧ integers>=0.3448	∧ yieldpoints>=0.0143			
(51/ 14) list ← bbLen>=7	∧ systems<=0.0741	∧ loads>=0.3	∧ systems<=0.0465	∧ peis<=0.2			
(39/ 8) list ← bbLen>=5	∧ stores<=0.1562	∧ loads>=0.3	∧ integers>=0.3529	∧ gcpoints<=0.1818	∧ peis>=0.1667		
(33/ 7) list ← bbLen>=5	∧ stores<=0.1562	∧ loads>=0.3	∧ integers>=0.3529	∧ peis<=0.15	∧ peis>=0.125	∧ calls<=0	
(25/ 3) list ← bbLen>=5	∧ stores<=0.12	∧ loads>=0.2222	∧ integers>=0.3889	∧ stores>=0.1	∧ branches>=0.1111		
(18/ 5) list ← bbLen>=5	∧ stores<=0.1613	∧ loads>=0.2941	∧ integers>=0.3846	∧ calls>=0.0769	∧ stores>=0.1111		
(27476/1946) orig ←							

Figure 4: Induced Heuristic Generated By Ripper

Benchmarks	Description
`linpack`	A numerically intensive program used to measure floating point performance of computers
`power`	Power pricing system optimization problem solver
`bh`	Barnes and Hut N-body force computation algorithm
`voronoi`	Computes the voronoi diagram of a set of points recursively on the tree
`aes`	A program to test vectors from the NIST standard encryption tests
`scimark`	A program for scientific and numerical computation

Table 7: Characteristics of a set of benchmarks that benefit from scheduling.

tions (call, system, load, and store) are the most important features, with the rest offering some fine tuning. For example, the first if-then rule predicts that it is beneficial to schedule blocks consisting of 7 instructions or more, that have a small fraction of call and PEI instructions, but possibly a larger fraction of load and integer instructions. Note that for this training set a large percentage of blocks were predicted not to benefit from scheduling.

5. RELATED WORK

Instruction scheduling is a well-known problem with a developed literature. It is also known that optimal instruction scheduling for complex processors is NP-complete [9]. For brevity and focus we describe the works, being most closely related to ours in that they consider application of machine learning to compiler optimization problems.

Calder et al. [3] used supervised learning techniques, namely decision trees and neural networks, to induce static branch prediction heuristics. The prediction rates of their approach resulted in a miss rate of 20% as compared with the 25% miss rate obtained using the best existing hand-crafted heuristics at the time. Their learning methodology is similar to ours, but there are important differences. First, they had a rich set of hand-crafted heuristics from which to from features. On the other hand, we had no pre-existing heuristics to investigate from which draw features. Second, their technique made it inherently easy to determine a label for their training instances. The optimal choice for predicting a branch was easily obtained by instrumenting their benchmarks to observe each branch's most likely direction. We obtained our labels using a simplified model of our target processor, which is imprecise as previously mentioned. Because our measurements are imprecise, it is impossible to determine the optimal choice of whether to schedule or not to schedule.

Monsifrot et al. [13] use a classifier based on decision tree learning to determine which loops to unroll. Like in Calder et al. [3], there were many hand-coded heuristics from which to draw features. In contrast to our approach and Calder's, they obtain labels by using timing measurements from a real machine. For each loop, they measure the effect of unrolling and not unrolling that particular loop. If the effect of unrolling is beneficial above some threshold, they create a positive training example pertaining to the loop. If unrolling the loop causes a degradation in performance, a negative training example is generated from the loop.

A group of researchers at MIT used genetic algorithms to tune heuristic priority functions in three compiler optimizations [17]. They generated, at random, expressions for a priority function for a specific compiler optimization, and formed an initial population for a genetic algorithm. They performed crossovers and mutations by modifying the expressions with relational and/or real-valued functions of random expressions. They derived priority functions for these tasks: hyperblock selection, spilling in register allocation, and data prefetching. Their generated heuristics outperformed hand-crafted ones on an architectural simulator. (However, simply by producing 399 heuristics at random and choosing the best they were able to outperform the hand-crafted heuristics.) Iterating the genetic programming produced a significantly better result only for the spilling priority function in register allocation, and it stabilized to the best performing genomes in a few iterations. Unsupervised learning, such as genetic programming, has two advantages over our technique: in the learning process it uses measured rather than simulated execution times, and it does not require a timing simulator. However, unsupervised learning is typically more complex, and the resulting functions are often more opaque. Also, this genetic programming work took days of CPU time to derive a heuristic, whereas our supervised learning procedure completes in seconds (once we have developed the training instances).

Cooper et al. [6] use genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding "good" compiler optimization sequences that reduced code size. Unfortunately, their technique is application-specific. That is, a genetic algorithm has to retrain for each program to decide the best optimization sequence for that program. The genetic algorithm builds up chromosomes pertaining to different sequences of optimizations and adapts these for each individual program. Mutations can involve adding new optimizations into the sequence or removing existing ones from the sequence. Their technique was successful at reducing code size by as much as 40%.

We previously reported [14] results on generating a priority function in instruction scheduling. Using supervised learning, we generated preference functions that determined the preferred instruction to schedule next from a pair of instructions (in the LS algorithm). Our conclusion was that machine learning could find, automatically, quite competent priority functions for local instruction scheduling heuristics. In later work [12, 11] we had some success applying reinforcement learning to the same problem.

6. CONCLUSIONS

Choosing when to apply potentially costly compiler optimizations is an important open problem. We consider here the particular case of instruction scheduling, with the possible choices being a traditional list scheduler (LS) and no scheduling (NS). Since many blocks do not benefit from scheduling, one can obtain most of the benefit of scheduling by applying it to a subset of the blocks. What we demonstrated here is that it is possible to induce a function that is competent at making this choice: we obtain almost all the benefit of LS at less than 1/4 of the cost.

On the way to this result we found that it helped to reduce noise: to remove training instances whose cost under different schedulers is within a chosen threshold value, i.e., not different enough to provide a good "signal" on which to train. Interestingly, this instance filtering improved *both* the efficiency and the effectiveness of our induced function.

Sometimes (perhaps only rarely) it is beneficial to perform instruction scheduling in a JIT, depending on how long the program runs, etc. If it is rarely worthwhile, that only emphasizes the need for our heuristic to decide when to apply it. The general approach we took here should apply in other JIT situations. Of course all we have demonstrated rigorously is that it works for one Java compilation system.

We found supervised learning to work excellently for this task. Thus, beyond achieving good performance on the task, we obtain the additional benefits of a simple cheap learning algorithm that produces understandable heuristics. As with any machine learning technique, devising the appropriate features is critical. Choosing whether to apply an instruction scheduler turns out to require only simple, cheap-to-compute features. More complex compiler optimizations, such as redundancy elimination, almost certainly need more complex features to use in deciding if the optimization is likely to be worthwhile, but we hope that this positive experience will inspire success on harder problems.

7. ACKNOWLEDGMENTS

We thank Amy McGovern for coding the simulator and for prior work in this domain. This material is based upon work supported by the National Science Foundation under grant number CCR-0085792. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), Feb. 2000.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[4] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. Rau, and M.Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 58–67, 1992.

[5] W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, Lake Tahoe, CA, Nov. 1995.

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, 1999.

[7] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale, Feb. 1985.

[8] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, July 1981.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, CA, 1979.

[10] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings ACM SIGPLAN '86 Conference on Programming Language Design and Implementation*, pages 11–16, 1986.

[11] A. McGovern, E. Moss, and A. G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Machine Learning*, 49(2/3):141–160, 2002.

[12] A. McGovern and J. E. B. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In S. Solla, editor, *Advances in Neural Information Processing Systems 11*, Cambridge, MA, 1998. MIT Press.

[13] A. Monsifrot and F. Bodin. A machine learning approach to automatic production of compiler heuristics. In *Tenth International Conference on Artificial Intelligence: Methodology, Systems, Applications, AIMSA*, pages 41–50, September 2002.

[14] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanović, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Proceedings of Neural Information Processing Systems 1997 (NIPS*97)*, Denver CO, Dec. 1997.

[15] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1997.

[16] Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM98 Benchmarks*, 1998.

[17] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, Ca, June 2003. Association of Computing Machinery.

[18] C. Young and M. Smith. Better global scheduling using path profiles. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 199–206, Nov. 1995.