Hybrid Optimizations: Which Optimization Algorithm to Use?

John Cavazos[1], J. Eliot B. Moss[2], and Michael F.P. O'Boyle[1]

[1] Member of HiPEAC
Institute for Computing Systems Architecture (ICSA)
School of Informatics, University of Edinburgh
Edinburgh, UK
[2] Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA USA

Abstract. We introduce a new class of compiler heuristics: *hybrid optimizations*. Hybrid optimizations choose dynamically at compile time which optimization algorithm to apply from a set of different algorithms that implement the same optimization. They use a *heuristic* to predict the most appropriate algorithm for each piece of code being optimized. Specifically, we construct a hybrid register allocator that chooses between linear scan and graph coloring register allocation. Linear scan is more efficient, but sometimes less effective; graph coloring is generally more expensive, but sometimes more effective. Our setting is Java JIT compilation, which makes optimization algorithm efficiency particularly important.
Our hybrid allocator decides, based on features of a method, which algorithm to apply to that method. We used supervised learning to induce the decision heuristic. We evalute our technique within Jikes RVM [1] and show on average it outperforms graph coloring by 9% and linear scan by 3% for a typical compilation scenario. To our knowledge, this is the first time anyone has used heuristics induced by machine learning to select between different optimization algorithms.

1 Introduction

Compiler writers constantly invent new optimization algorithms to improve the state of the art. They frequently arrive at significantly different algorithms for a particular compiler optimization. Often, however, there is no clear winner among the different algorithms. Each algorithm has situations in which it is preferable to the other algorithms. For example, many different register allocation algorithms have been invented that achieve either a good running time performance [8, 6, 10, 2], possibly at the expense of increased allocation time, or a reduction in allocation time [13, 16] at the expense of performance. Two register allocation algorithms that differ in these seemingly mutually exclusive goals are graph coloring and linear scan.

Graph coloring is an aggressive technique for allocating registers, but is computationally expensive due to its use of the interference graph, which can have a worst-case size that is quadratic in the number of live ranges. Linear scan (LS), on the other hand, does not build an interference graph, but instead allocates registers to variables in a

greedy fashion by scanning all the live ranges in a single pass. It is simple, efficient, and produces a relative good packing of all the variables of a method into the available physical registers. Graph coloring can sometimes lead to more effective packing of the registers, but it can be much more expensive than linear scan.[1]

We invent a new class of optimization heuristics, *hybrid optimizations*. Hybrid optimizations assume that one has implemented two or more algorithms for the same optimization. A hybrid optimization uses a heuristic to choose the best of these algorithms to apply in a given situation. Here we construct a hybrid register allocator that chooses between two different register allocation algorithms, graph coloring and linear scan. The goal is to create an allocator that achieves a good balance between two factors: trying to find a good packing of the variables to registers (and thereby achieving good running time performance) and trying to reduce the overhead of the allocator.

We discuss how we use supervised learning to construct a hybrid allocator. The induced heuristic should be significantly cheaper to apply than register allocation itself; thus we restrict ourselves to using properties (features) of a method that are cheap to compute or have already been computed in a previous compilation phase.

The contributions of the paper are:

1. To our knowledge, this is the first time anyone has used properties of code to construct automatically (using machine learning) a heuristic function that selects between different optimization algorithms.
2. We can construct an allocator that is as good as always using graph coloring with a significant reduction in allocation/compilation time.

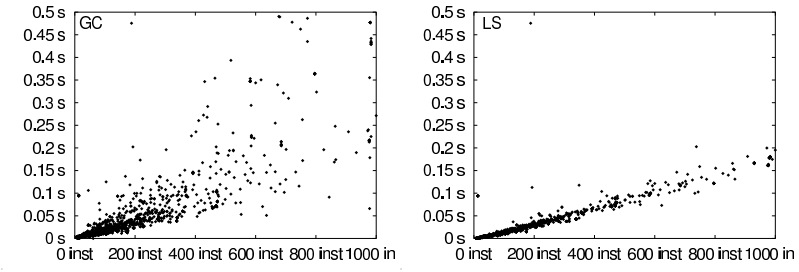


Fig. 1. Algorithm running time versus method size for GC and LS. Each point represents a method, where x-axis is size of method and y-axis is time to allocate.

2 Motivation

We measured the time to run GC and LS for the case of 24 registers (12 volatile (callee-save) registers and 12 non-volatile (caller-save) registers) on a PowerPC processor. Fig-

[1] Poletto et al. [13] show data where graph coloring can be as much as 1000 times more expensive than linear scan as the size of the live variables grows.

ure 1 shows scatter plots of the running time versus method size (expressed in number of instructions). Both graphs omit outliers that have very large running times. LS's running time is consistent and fairly linear. GC's time is in the same general region, but is worse and does not have as clear a linear relationship with method size (this is to be expected, given the overhead of constructing an interference graph in the GC algorithm).

Algorithm	(μs) / Inst	(μs) / Inst Without Outliers
GC	1241	300
LS	183	176

Table 1. Average running time to allocate with GC and LS in microseconds (μs) per instruction.

Table 1 gives statistics on the measured running times of GC and LS. The second column gives the average time to allocate registers in units of microseconds per (low-level intermediate code) instruction. GC is nearly 7 times slower than LS on this measure. However, if we exclude a small number of methods that took more than 1 second to schedule (outliers), then the ratio is 1.7. This shows that a small percentage of methods strongly bias GC's average running time. A possible strategy to ameliorate this is to predict when applying GC will benefit a method over LS, and run GC only in those cases. This is exactly the goal of a hybrid optimization. A hybrid optimization will reduce compilation effort, using an efficient algorithm most of the time, but will use a more effective, but expensive, optimization algorithm seldomly, when it deems the additional benefit is worth the effort. This trade-off is especially interesting if compilation time is important, such as in a JIT (just-in-time) compilation environment.

3 Problem and Approach

We want to construct a heuristic that with high effectiveness predicts which allocation algorithm is most beneficial to apply. We opted not to construct a hybrid allocator by hand, but instead to try to induce a choice function automatically using *supervised learning* techniques.

Developing and fine-tuning a hybrid allocator manually requires experimenting with different features (i.e., combinations of features of the method). 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 hybrid allocator that is based on the features of the method. Rule induction heuristics are typically faster to induce than using other machine learning techniques, [2] they are more understandable than heuristics learned using other techniques (e.g., neural networks and decision trees), and are easier to make work (than unsupervised learning).

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

The first step in applying supervised learning to this problem requires phrasing the problem as a classification problem. For this task, this means that each method is represented by a training instance and each training instance is labeled with respect to whether graph coloring achieves enough additional benefit (fewer spills) over linear scan to warrant applying it.

3.1 Features

What properties of a method might predict which allocation algorithm to use? One can imagine that certain properties of a method's interference graph might predict whether or not to use graph coloring. However, building the interference graph often dominates the overall running time of the graph coloring algorithm. Since we require cheap-to-compute features, we specifically choose not to use properties of the interference graph.

Instead, we use features that have previously been computed for other compilation phases. For instance, we use features summarizing the control flow graph, *CFG*, such as statistics pertaining to regular (non-exceptional) versus exceptional edges.[3] We also use features that describe liveness information, such as the number of variables that are live in and out of a block. We also try other cheap-to-compute features that we thought might be relevant. Computing these features requires a single pass over the method.

Features	Meaning
In/Out Edges	Number of CFG In/Out Edges
Exception In/Out	Number of CFG Exceptional In/Out Edges
Live on Entry/Exit	Number of edges live on entry/exit
Intervals	Number of live intervals
Virtual Registers	Number of virtual registers
Insts per block	Number of instructions per blocks
Insts per method	Number of instructions in method
Blocks per method	Number of blocks in method

Table 2. Features of a method.

The features can be grouped into three different categories. The first set of features pertains to edges in the control flow graph. These features include regular CFG edges and exceptional CFG edges. The second set of features pertains to the live intervals. We provide features for statistics describing the number of intervals (roughly, "variables") that are live going in and out of the blocks. This set also includes features for the number of live intervals and virtual registers. [4] The third set of features describes statistics about sizes of blocks and the total number of instructions and blocks in the method. See Table 2 for a complete list of the features. These features were either pre-computed or

[3] Exceptional edges represent control flow if an exception is raised.

[4] Virtual registers refers to user-defined variables and compiler generated temporaries that we would like to allocate to machine registers.

as cheap to compute as we can imagine while offering some useful information. These features work well so we decided not to refine them further. Our domain knowledge allowed us to develop a set of features that on our first attempt produced highly-predictive heuristics.

We present all of the features (except number of instructions and blocks) in several forms, such as minimum, maximum, total, and mean. This allows the learning algorithm to generalize over many different method sizes.

It might be possible that a smaller set of features would perform nearly as well. However, calculating features and evaluating the heuristic functions takes less than 0.1% of compile time (a negligible fraction of compile time) in our experiments, so we did not explore this possibility. In addition, rule induction automatically finds the set of most significant features, so we did not need to eliminate features manually. One final observation is that these features are machine independent and should be useful across a wide range of systems. For instance, we performed experiments varying the number of available registers and found this set of features worked well across all configurations we evaluated.[5]

3.2 Generating Training Instances

We have constructed a set of features and now take the next step, generating training instances. Each training instance consists of a vector of feature values, plus a boolean classification label, i.e., *LS* (Linear Scan) or *GC* (Graph Coloring), depending on which algorithm is best for the method.

After the Java system compiles and optimizes each Java method, the last phase involves presenting the method for register allocation. As we allocate the variables in the method to registers we can compute the features in Table 2. We instrument both a graph coloring allocator and a linear scan allocator to print into a trace file, for each method, raw data for forming a training instance.

Each raw datum consists of the features of the method that make up the training instance and statistics used to calculate the label for that instance. For the particular step of computing the features of a method, we can use either algorithm since the features are not algorithm specific. However, computing the final statistics used for labeling requires allocating the method with both graph coloring and linear scan. These statistics include the time to allocate the method with each allocation algorithm, and the number of additional spills incurred by each algorithm. We discuss the use of these statistics for labeling each training instance in Section 3.3.

We obtain the number of spills for a method by counting the number of loads and stores added to each block after register allocation and multiplying this by the number of times each basic block executes. We obtain basic block execution counts by profiling the application. We emphasize that these steps take place in an instrumented compiler, run over a benchmark suite, and all happen *off-line*. Only the heuristics produced by supervised learning are part of the production compiler and these heuristics are fast.

[5] The exact heuristic *functions* may vary. For rather larger registers (24 or more), graph coloring only rarely beats linear scan, while for small sets (4 or 8) graph coloring is even more important than the medium size case we consider here (12).

3.3 Labeling Training Instances

We label an instance based on two different thresholds, a *cost threshold* and a *benefit threshold*. The cost threshold pertains to the time it takes to allocate registers with each algorithm. The benefit threshold pertains to the number of spill loads and stores incurred by each allocation algorithm. We use spills as an indirect method to measure the benefit of an allocation algorithm instead of using the direct metric of a method's running time. Measuring running time of an individual method is hard to do reliably, whereas measuring dynamic spill count is easy to do and does not change between runs. Our results show that this indirect metric works well.

For the experiments in this paper we use the following procedure to label the training instances. We label an instance with "GC" (prefer graph coloring) if the number of spills using linear scan minus the number of spills using graph coloring on the same method is greater than some threshold (Spill_Threshold). We label an instance with "LS" (prefer linear scan) if there is no spill benefit by allocating the method with graph coloring. We also label an instance with "LS" if the cost of using graph coloring is above a threshold (Cost_Threshold) more than the cost of applying linear scan. Those instances where there is no clear benefit, in terms of spills or cost, in using graph coloring or linear scan are discarded and not considered for learning. They do not provide useful guidance and only push rule induction to try to make inconsquential fine distinctions. Figure 2 depicts this algorithm for labeling.

```
if (LS_Spill − GC_Spill > Spill_Threshold)
    Label as GC;
else if (LS_Spill - GC_Spill <= 0)
    Label as LS;
else if (LS_Cost/GC_Cost > Cost_Threshold)
    Label as LS;
else
    { // No Label (discard instance) }
```

Fig. 2. Procedure for labeling instances with GC and LS

We experimented with different threshold values for the spill benefit and cost threshold. Varying these threshold values gave us a variety of different heuristic functions. We report results for the best heuristic function found from 6 heuristic functions explored.

3.4 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 [9].

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 in Section 3.5.

3.5 A Sample Induced (Learned) Heuristic

As we mentioned, rule sets are easier to comprehend and are often compact. It is also relatively easy to generate code from a rule set that can be used to build a hybrid allocator.

Table 3 shows a rule set induced by our training data. If the right hand side condition of any rule (except the last) is met, then we will apply GC on the method; otherwise the hybrid allocator predicts that GC will not benefit the method and it applies LS. The numbers in the first two columns give the number of training examples that are correctly and incorrectly classified by the rule.

```
(   20/ 9) GC ← avgLiveOnExitBB >= 3.8  ∧ avgVirtualRegsBB >= 13
(   22/13) GC ← avgLiveOnEntryBB >= 4   ∧ avgCFGInEdgesBB >= 1.4  ∧ avgLiveOnExitBB >= 5.5 ∧
                numberInsts <= 294
(   10/ 5) GC ← avgLiveOnExitBB >= 4.3  ∧ maxLiveOnEntry <= 13
(   12/ 2) GC ← avgLiveOnExitBB >= 3.7  ∧ maxLiveOnEntry >= 9       ∧ numVirtualRegs >= 895 ∧
                maxLiveIntervals >= 38  ∧ maxLiveIntervals <= 69
(1815/78)  LS ←
```

Table 3. Induced Heuristic Generated By Ripper.

In this case we see that liveness information and the number of virtual registers 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 use graph coloring on methods consisting blocks with a high average number of live intervals exiting the block and a high average number of virtual registers in the block. Note that for this training set a large percentage of methods ($1815+78=1893$ of $1986=95\%$) were predicted not to benefit from graph coloring. As we see in Section 6.2, determining the feature values and then evaluating rules like this sample one does not add very much to compilation time, and takes much less time than actually allocating the methods.

3.6 Integration of the Induced Heuristic

After training, the next step is installing the heuristic function in the compiler and applying it *online*. When the optimizer invokes register allocation on a method, we first compute our features for the method. We then call the heuristic function, passing the features. If the heuristic functions says to use graph coloring, we do so, otherwise we apply linear scan. If a method is re-optimized, as sometimes happens in adaptive compilation, the heuristic will be applied again, to the new code, and may make a different decision.

We include in our reported timings of allocation the cost of computing features and invoking the heuristic function. These costs are quite small compared with running the allocation algorithms.

4 Experimental infrastructure

We implemented our register allocation algorithms in Jikes RVM, a Java virtual machine with JIT compilers, provided by IBM Research [1]. The system provides a linear scan allocator in its optimizing compiler. In addition, we implemented a Briggs-style graph coloring register allocator [5].

We experiment with Jikes RVM in an adaptive scenario, which is a typical scenario for compiling Java programs. In this scenario, the compiler identifies and optimizes only frequently executed (*hot*) methods at progressively higher levels of optimization [3]. This is the typical compilation scenario of a Java JIT compiler. It achieves most of the benefit of optimizing all methods, but with much lower compilation cost.

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 whose microarchitecture is representative of the state of the art in processor implementations. We motivated this work allocating 24 registers, the number available on a PowerPC. We present results for allocating 12 registers (6 volatile and 6 non-volatile), which corresponds to AMD, ARM, and many embedded processors in wide-spread use.

4.1 Benchmarks

We examine 7 programs drawn from the SPECjvm98 suite [14] for the experiments in this paper. We detail these benchmarks in Table 4. We ran these benchmarks with the largest data set size (called 100).

Program	**Description**
`compress`	Java version of 129.compress from SPEC 95
`jess`	Java expert system shell
`db`	Builds and operates on an in-memory database
`javac`	Java source to bytecode compiler in JDK 1.0.2
`mpegaudio`	Decodes an MPEG-3 audio file
`raytrace`	A raytracer working on a scene with a dinosaur
`jack`	A Java parser generator with lexical analysis

Table 4. Characteristics of the SPECjvm98 benchmarks.

5 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. First, we envision developing and installing of the heuristic "at the factory", and it will then be applied to code it has not "seen" before. Second, 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 hybrid allocator on a benchmark, we consider three kinds of results: *spill loads*, *total running time*, and *benchmark running time*.

Spill loads refers to the additional number of loads (read memory accesses) incurred by the allocation algorithm. Spill loads give an indication of how well the allocator is able to perform its task. Memory accesses are expensive, and although the latency of some additional accesses can be hidden by overlapping with execution of other instructions, the number of spill loads is highly correlated with application running time. We do not count spill stores because their latency can be mostly hidden with *store buffers*. (A store buffer is an architectural feature that allows computation to continue while a store executes. Store buffers are typical in modern architectures.)

Total time refers to the running time of the program including compile time. We compare our hybrid allocator to always using linear scan and always using graph coloring. 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. (We also supply measurements of those costs in Section 6.2.)

Running time refers to the running time of the program without compilation time. This measures the change in execution time of the allocated code, compared with always using linear scan and always using graph coloring. This validates not only the heuristic function but also our instance labeling procedure, and by implication the spill model we used to develop the labels.

The goal of the hybrid allocator is to achieve application running time close to always applying graph coloring, while reducing compilation time to substantally less than applying graph coloring to every method. If these two objectives are met, a hybrid allocator should reduce total time compared to always applying either graph coloring or linear scan.

6 Experimental Results

We aimed to answer the following questions: How *effective* is the hybrid allocator in obtaining best application performance compared to graph coloring? How *efficient* is our hybrid allocator compared to allocating all methods with linear scan? We ask these questions on the SPECjvm98 benchmark suite. We then consider how much time it takes to apply the hybrid heuristic in the compiler.

We answer the first question by comparing the running time of the application, with compilation time removed. We address the second by comparing total time which includes time spent compiling including allocating. Since we are using adaptive optimization in a JIT compiler, *total time* is the key measure of overall importance.

We requested that the Java benchmark iterate 26 times garbage collecting between iterations. The first iteration will cause the program to be loaded, compiled, and allocated according to the allocation algorithm. The remaining 25 iterations should involve no compilation; we use the *best* of the 25 runs as our measure of application performance.

Section 6.1 discusses the benefit of our allocators with respect to reducing spills. Section 6.2 discusses the cost of using the hybrid heuristic. Finally, Section 6.3 discusses the effect of using hybrid allocators in reducing program execution (running and total) time.

6.1 Spill Loads

Before looking at execution times on an actual machine, we consider the quality of the induced hybrid allocator (compared with always applying either graph coloring or linear scan) in terms of the number of spill loads added by register allocation. Spill loads predict whether a method will benefit from the additional effort of applying the more expensive graph coloring allocator.

Spill loads are used in the labeling process, thus we hoped that our hybrid allocator would perform well on the spill load metric. Comparing the allocators against spill loads allows us to validate the learning methodology, independently of validating against the actual performance on the target machine.

We calculate the dynamic number of spill loads added to each method by multiplying the number of spill loads added to each block by the number of times that block is executed (as reported by profiling information). Then we sum the number of dynamic spill loads added to each block. We obtain the dynamic number of spills loads for the entire program by summing the number of dynamic spill loads added to each method. More precisely, the performance measure for program P is:

$$\text{SPILLS}_\pi(P) = \sum_{M \in P} \sum_{b \in M} (\#\text{ Executions of } b) \cdot (\text{spill loads added to } b \text{ under allocator } \pi)$$

where M is a method, b is a basic block in that method, and π is hybrid, graph coloring, or linear scan.

Table 5 shows the spill loads for each allocator as a ratio to spill loads produced by our graph coloring algorithm. These numbers are given as geometric means over the 7 SPECjvm98 benchmarks. We see improvements for the hybrid allocator over linear scan. These improvements do not correspond exactly to measured execution times, which is not surprising given that the number of spill loads is not an exact measure of performance on the architecture. What the numbers confirm is that the induced heuristic indeed improves the metric on which we based its training instances. Thus, supervised learning was able to solve this learning problem. Whether we get improvement on the

Register Allocator	jack	db	javac	mpeg-audio	ray-trace	com-press	jess	geo. Mean
HYBRID	1.07	1.00	1.04	1.49	1.05	2.01	1.33	1.25
LS	1.07	1.20	1.07	2.13	1.55	2.01	1.42	1.44

Table 5. Spill loads for hybrid and linear scan relative to graph coloring.

real machine is concerned with how predictive reducing spill loads (i.e., the spill model) is to benchmark performance. The hybrid allocator was able to reduce the number of spills obtained by linear scan by 19%. In Section 6.3, we see that reducing the number of dynamic spill loads leads to an average reduction in running time of 5% for hybrid allocation over linear scan for our set of benchmarks.

6.2 The Cost of Evaluating a Hybrid Heuristic

Table 6 gives a breakdown of the compilation costs of our system, and statistics concerning the percentage of methods and instructions allocated with each allocator. The costs are given as geometric means over the 7 benchmarks.

Allocator	GCM	GCI	LSM	LSI	f/a	f/c	a/c
GC	100.00%	100.00%	0.00%	0.00%	0.00%	0.00%	21.26%
HYBRID	2.42%	9.53%	97.58%	90.47%	1.20%	0.09%	7.31%
LS	0.00%	0.00%	100.00%	100.00%	0.00%	0.00%	3.41%

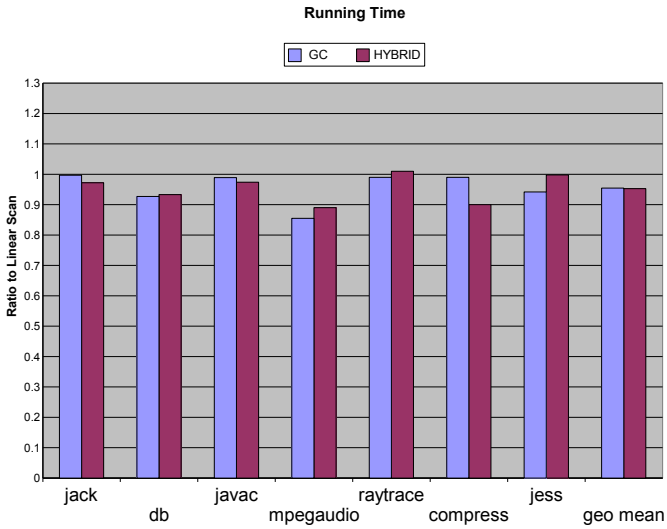
Table 6. Cost breakdowns: GCM/GCI = GC allocated methods/instructions; LSM/LSI = LS allocated methods/instructions; f = time to evaluate features and heuristic function; a = time spent allocating methods; c = compile time excluding allocation time.

Here are some interesting facts revealed in the table. First, the fraction of methods and of instructions allocated with GC drops significantly for our hybrid scheme. Second, the fraction of instructions allocated with GC, which tracks the relative cost of allocation fairly well, is about 4 times as big as the fraction of methods allocated with GC, implying that the hybrid allocator tend to use GC for longer methods. This makes sense in that longer methods probably tend to benefit more from graph coloring.

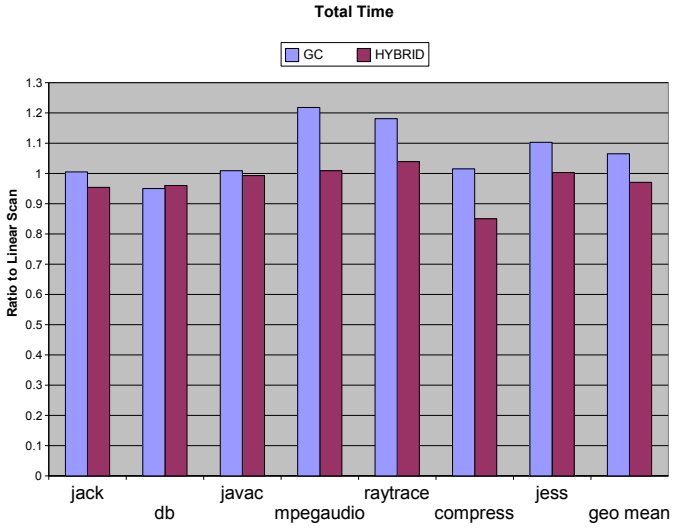
Third, the cost of calculating the heuristic, as a percentage of compilation time, is less than 0.1%. Finally, we obtain a factor of 3 reduction in allocation time compared with GC (applying graph coloring to every method).

6.3 Effectiveness and Efficiency

We now consider the quality of the hybrid heuristic induced for adaptive optimization. Figure 3(a) shows the impact of the hybrid allocation and GC on application running



(a) Running Time Using Hybrid versus Graph Coloring relative to Linear Scan



(b) Total Time Using Hybrid versus Graph Coloring relative to Linear Scan

Fig. 3. Efficiency and Effectiveness Using Hybrid Allocator with Adaptive Compiler.

time, presented relative to LS. Here there is little variation between graph coloring and hybrid allocation across the benchmarks, with GC doing well at 5% and the hybrid allocator doing just as well. Given the lower cost of the hybrid allocator to run, it is preferable to running GC all the time. The results are fairly consistent across the benchmarks, though some benchmarks improve more than others.

As Figure 3(a) shows we can effectively select dynamically the methods where graph coloring improves over linear scan with our hybrid heuristic. The hybrid allocator is effective, but what happened to efficiency?

Figure 3(b) shows the total time of the hybrid allocator and GC (always performing graph coloring), relative to LS (always performing linear scan). The average for the total time graph shows an improvement using the hybrid allocator over either always using linear scan or always using graph coloring. Using the hybrid allocator we can achieve up to an 15% improvement over linear scan for compress with an average improvement of 3%. Using graph coloring, we can also achieve a substantial improvement on some benchmarks over linear scan, but we also incur a significant degradation on some programs (up to 22% for mpegaudio) over linear scan. However, by selective applying GC only when it is beneficial we can reduce total time by 9% on average using hybrid allocator over graph coloring. The improvement in total time of hybrid allocation over graph coloring shows we were able to cut the compilation effort (and therefore total time) significantly.

7 Related Work

Lagoudakis et al. [11] describe an idea of using features to choose between algorithms for two different problems, order statistics selection and sorting. The authors used reinforcement learning to choose between different algorithms for each problem. For the order statistics selection problem, the authors choose between Deterministic Select and Heap Select. For sorting, they choose between Quicksort and Insertion Sort. The hybrid algorithms were able to outperform each individual algorithm.

Cavazos et al. [7] describe an idea of using supervised learning to control whether or not to apply instruction scheduling. They induced heuristics that used features of a basic block to predict whether scheduling would benefit that block. Using the induced heuristic, they were able to reduce scheduling effort by as much as 75% while still retaining about 92% of the effectiveness of scheduling all blocks.

Monsifrot et al. [12] use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC and an IA64, where their learned scheme showed modest improvement.

Stephenson et al. [15] used genetic programming to tune heuristic priority functions for three compiler optimizations within the Trimaran IMPACT compiler. For two optimizations they achieved significant improvements. However, these two pre-existing heuristics were not well implemented. For instance, turning off data prefetching completely is preferable and reduces many of their significant gains. For the third optimization, register allocation, they were able to achieve on average only a 2% improvement over the manually tuned heuristic.

Bernstein et al. [4] describe an idea of using three heuristics for choosing the next variable to spill, and choosing the best heuristic with respect to a cost function. This is similar to our idea of using a hybrid allocator to choose which algorithm is best based on properties of the method being optimized. Their technique applies all the spill heuristics and measures the resultant code with the cost function. Their technique results in about a 10% reduction in spills and a 3% improvement in running time. Our technique, on the other hand, does not try each option, but instead uses features of the code to make a prediction. By making a prediction using simple properties of the code, our heuristics are more efficient while still remaining effective. In fact, our technique could be used as an alternative to the cost function used in their work.

8 Conclusions

Choosing *which* optimization algorithm to apply among different optimization algorithms that differ in efficiency and effectiveness can avoid potentially costly compiler optimizations. It is an important open problem. We consider here the particular case of register allocation, with the possible choices being linear scan, graph coloring , and a hybrid allocator that chooses between these two algorithms. Since many methods do not gain additional benefit from applying graph coloring over linear scan, a hybrid allocator applies graph coloring only to a subset of the methods. We demonstrated that for an aggressive optimizing compiler (in a typical adaptive scenario) it is possible to induce a function automatically that is competent at making this choice: we obtain the effectiveness benefit of graph coloring and the efficiency benefit of linear scan.

Sometimes (only rarely) it is beneficial to perform graph coloring in a JIT, depending on how long the program runs, etc. Since it is only rarely worthwhile, that emphasizes the need for our heuristic to decide when to apply it. The general approach we took here should apply in other optimization situations.

Supervised learning worked well for this learning task. In addition, the learning algorithm we use produces understandable heuristics. As with any machine learning technique, devising the appropriate features is critical. Choosing which register allocation algorithm to apply turns out to require only simple, cheap-to-compute features.

We conclude that machine learning shows promise for developing heuristics for choosing between multiple algorithms for the same optimization task. A useful direction for future exploration is more expensive, rarely helpful, yet sometimes essential, optimizations, such as redundant load and store elimination.

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):211–238, Feb. 2000.
2. A. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and*

Implementation, pages 243–253, Snowbird, Utah, June 2001. Association of Computing Machinery.

3. M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Minneapolis, MN, Oct. 2000. ACM Press.
4. D. Bernstein, D. Q. Goldin, M. C. Golumbic, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 258–263, Portland, Oregon, June 1989. ACM Press.
5. P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 275–284. ACM Press, June 1989.
6. P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.
7. J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 183–194, Washington, D.C., June 2004. ACM Press.
8. G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN '82 Conference on Programming Language Design and Implementation*, pages 98–101, Boston, Massachusetts, June 1982. ACM Press.
9. W. W. Cohen. Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 115–123, Lake Tahoe, CA, Nov. 1995. Morgan Kaufmann.
10. L. George and A. W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.
11. M. G. Lagoudakis and M. L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518, Stanford, CA, June 2000. Morgan Kaufmann.
12. A. Monsifrot and F. Bodin. A machine learning approach to automatic production of compiler heuristics. In *Tenth International Conference on Artifi cial Intelligence: Methodology, Systems, Applications (AIMSA)*, pages 41–50, Varna, Bulgaria, September 2002. Springer Verlag.
13. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.
14. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA. *SPEC JVM98 Benchmarks*, 1998.
15. 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*, pages 77–90, San Diego, Ca, June 2003. ACM Press.
16. O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 142–151, Montreal, Canada, June 1998. ACM Press.