Application of Machine Learning to Inlining Decisions

308-761A

Ondřej Lhoták

December 13, 2001

1 Introduction

Object-oriented languages such as Java encourage programmers to separate their programs into very short functions, incurring a high run-time overhead due to the frequent function calls. This overhead can be reduced by inlining appropriate function call sites. However, inlining can also increase code size, so inlining sites must be carefully selected.

Inlining has secondary effects, which are poorly understood but sometimes significant to performance. For example, an inlined call site may provide opportunities for an optimizing compiler to perform optimizations which it would otherwise not be able to perform. Also, inlined functions may open up opportunities for further inlining. On the other hand, even a modest increase in code size may have hard-to-predict effects on performance caused by increased pressure on the instruction cache of the processor. Also, inlining may have unpredictable effects on register utilization. An algorithm considering these effects could potentially provide better inlining decisions than the heuristics currently in use. Because the effects are poorly understood, machine learning appears to be a promising approach. Also, because the profitability of inlining depends on the other optimizations that a compiler supports, a generic machine learning approach could be used to create heuristics specific to particular compilers.

2 Existing Work

I am not aware of any published applications of machine learning to inlining decisions. Machine learning has been applied to static branch prediction [2]. A summary of the possible benefits and drawbacks of inlining, as well as commonly used heuristics, is given in [5]. Several papers on heuristics for inlining decisions, some of them quite sophisticated, have been published. A good summary of the shortcomings of common inlining decision heuristics appears in [6]. An experimental study of various heuristics applied to Java is given in [1]. The inlining decision algorithm presented in [3] takes into account estimates of the potential for compiler optimization opportunities created by inlining. The Soot bytecode optimization framework is described and experimentally evaluated in [7]. This includes a description and evaluation of the inlining heuristics implemented in Soot.

3 The Learner

The goal of the learner is to construct an oracle which, for each inlinable call site in a program being compiled, decides whether or not that site should be inlined.

The information about each call site is summarized in ten real-valued attributes of the site, which are believed to be relevant to inlining decisions:

Size of function to be inlined (henceforth callee)
Many inlining heuristics only allow short callees to be inlined. For example, the standard heuristic used by Soot inlines only callees consisting of at most twenty statements of Jimple, the intermediate representation used in Soot. In this project, the size of the callee is measured as the logarithm of the number of Jimple statements.

Size of function containing the call site (henceforth caller)
Some inlining heuristics only allow inlining into callers smaller than a certain threshold. The

standard heuristic used by Soot inlines only into callers consisting of at most 5000 statements of Jimple. In this project, the size of the caller is measured as the logarithm of the number of Jimple statements.

Expansion factor of caller and callee due to inlining

Many inlining heuristics consider the amount of code growth caused by inlining a site, compared to the original size of the uninlined code for the caller and callee. The standard Soot heuristic inlines call sites where the expansion factor is at most three. The expansion factor is defined as $\frac{e+r}{r}$, where e is the number of Jimple statements in the callee, and r is the number of Jimple statements in the caller.

Number of arguments to callee

The arguments to a method call must be passed when calling it, so their number affects the overhead of the call which is eliminated by inlining.

Number of constants passed as arguments

When compile-time constants are passed as arguments to a method, the method can be specialized to those arguments when it is inlined, possibly creating additional optimization opportunities. Intra-procedural constant propagation is done before counting the number of compile-time constants which are passed in the call.

Number of places in the program where callee is called

If a method is called in only a small number of places in the program, it can be inlined at all the call sites with little code growth. Because this number has such a wide range, its logarithm rather than the actual number is used as input to the learner.

Number of call sites in callee

This is another measure of the size of the callee. The learner uses the logarithm of the number of call sites in the callee.

Number of call sites in caller

This is another measure of the size of the caller. The learner uses the logarithm of the number of call sites in the caller.

Number of live local variables before call site

Inlining a call site gives the JIT compiler in the virtual machine more flexibility in allocating registers to the values used in the caller and callee. The number of local variables which are live before the call site affects the number of values which need to be allocated to the limited number of registers. Because this number has such a wide range, its logarithm rather than the actual number is used as input to the learner.

Loop nesting level of call site

Call sites within loops are likely to be called more times than call sites which are not in loops, so their cumulative method call overhead is likely to be higher. As Muchnick [5] points out, it is not always possible to determine the loop nesting level in the source code from the bytecode (which is the input to Soot). Therefore, the loop nesting level is approximated by the number of targets of back edges crossing the call site, which in most typical Java methods is equal to the loop nesting level [4].

The function to be learned, is represented by $f : \mathbb{R}^{10} \to \mathbb{R}$, mapping the attributes of each call site to a real number representing the logarithm of the speedup expected from inlining the call site. The learner approximates the function as:

$$\hat{f}(\mathbf{x}) = w_c + \sum_{a \in \text{attributes}} (w_{a1} x_a + w_{a2} x_a^2)$$

where x_a is the value of attribute a for this call site, and the w's are the twenty-one weights which the learner is to learn. A site is considered profitable to inline if the speedup from inlining it as at least one (its logarithm is non-negative).

The algorithm used for learning is based on the gradient descent update rule

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E[\mathbf{w}](\mathbf{x})$$

where α is the learning rate, and $\nabla E[\mathbf{w}](\mathbf{x})$ is the gradient of the error function at the attribute vector $\mathbf{x}$. The error is measured by measuring the logarithm of the speedup of the program being optimized from inlining the call site, and subtracting it from the logarithm of the speedup as estimated by the function being learned, $\hat{f}(\mathbf{x})$. To minimize the effect of other processes running on the machine, the median of three measurements is used each time the runtime of the program is measured. This should eliminate most spurious speedups or slowdowns caused by external factors.

In order to measure the speedup from inlining a call site in a realistic environment, we must inline not only that one call site, but also the other call sites in the program, using an inlining heuristic similar to the function being learned. For this reason, the learning algorithm uses an ϵ-greedy strategy to decide which call sites to inline. For each trial, the current estimate of the inlining function is used to inline the sites for which inlining is deemed to be profitable, but with a probability of ϵ for each site, the estimate is flipped, and the site is inlined if and only if it is *not* profitable to inline it. In all experiments done for this project, the value of ϵ used was 0.125. Using this strategy, we get a program inlined approximately according to the current estimate of profitability, but with enough variation in the sites that are inlined to allow the learner to explore the effect of inlining sites that may not seem profitable to inline.

The running time of each trial is compared to the running time of the preceding trial to measure the speedup from inlining and uninlining the sites which were inlined and uninlined between the two trials. The expected speedup from the sites that were changed must be estimated using the function being learned so that the weights can be updated. This function, however, only provides the speedup for a single call site, so the values from the multiple call sites which were changed must somehow be combined. It may seem logical at first to simply add all the values, since they represent logarithms of speedup ratios. However, this would overestimate the combined speedup. If each of two call sites speeds the program up by a factor of two when inlined individually, inlining both of them is likely to speed up the program by much less than four-fold, due to Amdahl's Law. Unfortunately, the chosen representation of the program as attributes of individual call sites gives us no information about the relationships between call sites, and therefore the combined effect of inlining a number of them. We can use the average of the estimates instead of the sum to get an underestimate of the speedup, rather than an overestimate. An underestimate is more desirable, because it avoids the exponential explosion of the weights toward infinite values that the overestimate causes. The combined effect is therefore estimated by the average of the estimated speedups from all the sites that changed between two successive trials.

Additional measures need to be taken to prevent exponential explosion of weights. Specifically, the learning rate, α, needs to be kept low (at 0.01), and even lower (at 0.001), for updating the weights corresponding to squares of the attributes, as these can have larger values.

4 Other Learner Designs

In this section, I describe other learners which I tried before choosing the learner described in the preceding section.

The first learner modelled the problem as a Markov decision process. Its state was the program being inlined, with some subset of its inlinable call sites inlined. Possible actions were inlining any particular call site, and a stop action, indicating that all the sites that the learner had wanted to inline were already inlined. The state was described to the learner as the code size growth ratio compared to the original program with no sites inlined. Actions were described by the ten attributes presented in the previous section. The action value Q function was approximated by a CMAC, in which each attribute was discretized into ten possible tiles, and a sum of the trained weights of the relevant tiles, added to the state (growth) multiplied by a trained weight, was taken as an approximation to Q. At each step, exactly one call site was chosen ϵ-greedily to be inlined, and the Q value updated according to the Sarsa learning algorithm based on the speedup from that call site. The main problem with this approach was its slowness: because only one site was inlined at a time, the learner was not even able to observe all the call sites. Also, representing the state as the growth of the program violated the Markov assumption, since it is the specific set of sites that were inlined that influences running time, rather than the total amount of growth of the program. Additionally, the discretization of the attributes made it difficult for the learner to generalize from the limited number of observed call sites.

My other attempt at a learner was motivated by the exponentially exploding weights in the gradient descent learner. Believing that too many unobserved factors influence the speedup due to inlining any specific call site, I decided to not try to estimate the actual speedup from the call site, but simply to learn some function which would be positive if the call site was profitable to inline, and negative otherwise. Instead of updating the weights based on the error between the predicted speedup and the actual speedup, I updated them based only on the actual speedup. This avoided the exponential explosion in the weights, and seemed to perform comparably to the gradient descent learner in preliminary tests. However, this learner had no theoretical basis to suggest convergence, because it had no specific definition of a function to converge to. I therefore abandoned it in favour of the gradient descent learner once I tuned the latter to prevent its weights from exploding exponentially.

5 Experiment Design

The learning algorithm was tested on three benchmark programs, matrix and illness from the Ashes benchmark suite, and compress from the SpecJVM benchmark suite. These benchmarks were selected for their moderate number of inlinable call sites, because they were small enough to quickly analyze using Soot, and because they were known to benefit from inlining using existing heuristics. Benchmarks with too few inlinable call sites would provide little information to learn from, while benchmarks with too many inlinable call sites would make it difficult for the learner to find the significant ones in the limited time that the experiment runs. The following table summarizes the benchmarks:

Benchmark	Number of inlinable sites	Speedup using Soot inlining heuristic
matrix	37	1.43
illness	109	1.18
compress	60	1.23

A three-fold cross-validation study was performed, in which the learner was evaluated on each benchmark after learning for 500 trials on each of the other two benchmarks. For comparison, the learner was also evaluated on each benchmark after having learned for 500 trials on that same benchmark.

The experiment was conducted on shadow, a Sparc-based machine with 1GB of memory, using the Java virtual machine from the Sun JDK 1.2 with inlining by the JIT compiler disabled. It ran for approximately 48 hours.

6 Results

The results of the experiment are given in the following table, and in the plots at the end of this paper.

Benchmark	Original	Soot Optimized	Soot Inlined	Fully Inlined	Machine Learning (Cross)	Machine Learning (Same)	Best Encountered
Matrix	3.34	3.44	2.33	2.22	2.21	1.6	1.56
Illness	1.41	1.39	1.2	1.38	1.22	1.2	1.08
Compress	19.1	18.63	15.64	16.56	16.2	17.92	15.07

The table gives the running times in seconds of each benchmark after inlining using various inlining decision makers. The first column is the name of the benchmark. The second column is the running time of the original benchmark, with no optimization. The third column is the running time of the benchmark after it has been optimized using the whole-program optimizations of Soot, but with inlining turned off. The fourth column is the running time after the benchmark has been inlined using the heuristic in Soot, and optimized with Soot. The fifth column is the running time after all call sites that could be inlined have been inlined, and optimized with Soot. The sixth column is the running time after the benchmark has been inlined using a learner trained on the other two benchmarks, and optimized with Soot. The seventh column is the running time after the benchmark has been inlined using a learner trained on that same benchmark, and optimized with Soot. Finally, the eighth column is the shortest running time ever encountered on that benchmark during training. Because training is done on many randomly selected sets of sites to be inlined, this is likely to be a reasonable approximation to the performance of the best possible inlining strategy. The numbers in this table are presented in a bar graph at the end of this paper.

Also presented at the end of this paper are plots of the weights that were learned from training on the various combinations of the benchmarks, along with histograms of the distribution of attribute values in the benchmarks of the Ashes and SpecJVM benchmark suites. For each attribute a, the value $w_c + w_{a1}x_a + w_{a2}x_a^2$ (the contribution of the attribute to the estimated speedup) is plotted against x_a, the value of the attribute. The weights are plotted after training on each benchmark and each pair of benchmarks.

Finally, there are plots showing the error during training.

7 Interpretation of Results

The runtime measurements confirm that the optimizations performed by Soot benefit greatly from inlining. They also confirm that inlining is not always beneficial, since the fully inlined version is not the fastest for any of the three benchmarks. When it comes to the machine learning approach, the results are mixed.

On the matrix benchmark, both machine learning results are better than the heuristic used by Soot. The learner that learned on the matrix benchmark itself shows an overwhelming speedup over the Soot heuristic, and even the speedup for the cross-validation learner is significant. I am guessing that the learning algorithm found some small number of call sites which lead to a large improvement when inlined, and that the Soot heuristic was not able to find them for some reason. I have tried adjusting the parameters of the heuristic in Soot to try to match the performance of the learner, so far without success. I will take a closer look at the sites that the learner and Soot inlined to try to explain the large performance improvement, and hopefully improve the heuristic used by Soot.

On the illness benchmark, the learner matched the performance of Soot, with the cross-validation learner marginally slower. Slightly better running times occurred during training of the learner in a non-negligible number of the training runs. This seems to indicate that the learner was not able to express the set of call sites which would have to be inlined for this slight additional improvement.

On the compress benchmark, the learner was unable to match the performance of Soot, which was close to the best observed performance during training. Surprisingly, the cross-validation learner did significantly better than the learner trained on the compress benchmark itself. This could be caused by a number of reasons. Unlike the other two benchmarks, the compress benchmark has a large test harness, which probably has little effect on the running time, but contains a large number of call sites which may confuse the learner. The learner probably learned better on the other two benchmarks which did not have these confusing sites. Sadly, most real-world programs have large numbers of call sites, many of them having little effect on their running time, so the compress benchmark reflects real-world programs better than the other two.

The plots of the weights suggest that the learner was not able to express the correlation that there may be between the attributes that were measured and the effect of inlining the call sites. First, the constant weight w_c was in all cases very close to zero. If a good linear approximation to the function being learned exists, it is unlikely that its value should be zero precisely at the point where all the attribute values are zero, especially since many of the attributes are never zero for any call site. This suggests that the constant weight w_c was not sufficiently trained. Second, the slopes of the approximation are very different depending on the benchmark or benchmarks on which the learner was trained. In the arguments plot, for example, two of the curves suggest that sites with higher numbers of arguments provide a speedup, one suggests a slowdown, and three suggest no effect.

All of the plots are very linear, suggesting that perhaps the x_a^2 terms in the approximation were unnecessary. However, another plausible explanation is that the weights on the x_a^2 terms were not sufficiently trained, probably because the learning rate used for them was so low.

Finally, the plots of the error over time are very noisy, and show no sign of settling down. This could be due to the randomness introduced by the ϵ-greedy approach used to choose the training data, but it could also confirm the suspicion that the learner was not able to settle on a good approximation to the function.

8 Conclusion

The significant speedup on one of the benchmarks shows that a machine learning approach to inlining has a hope of improving on existing heuristics. However, the experiment also shows that the approach used in this study needs improvement before it will be useful on real-world programs. This is only a preliminary study; any study that could be considered significant would have to include a larger number of benchmarks bearing more similarity to real-world programs.

9 Future Work

The simplest way to extend this work would be to perform the same experiment on a wider range of benchmarks. However, I fear that this would only confirm the flaws in the current learner. The learner should be improved before a more significant study is done.

The disappointing results on the compress benchmark (and, to some extent, the illness benchmark) could be caused by any of many reasons, including:

1. The call site attributes observed may not correlate in general with the speedup due to inlining, and a machine learning approach based on these attributes therefore may have no hope of working.
2. The linear-quadratic approximation to the function may not be expressive enough to represent the true relationship between the call site attributes and the speedup.
3. The use of the average speedup may be an inappropriate way of combining the predicted speedup from inlining a number of call sites.
4. The gradient descent learner may not have had enough training to approximate the function well.

Little can be done to alleviate the first problem, short of exploring a completely different approach to inlining decision making. It is even difficult to determine whether the first problem was one of the actual causes of the poor performance.

The second problem could be explored by repeating the experiment using more expressive approximations. However, a more expressive approximation would be difficult to train with the small number of call sites available for training, and there would be a risk of overfitting.

The third problem could be explored by trying other ways to combine predicted speedup from multiple call sites. A more systematic approach would be to modify the algorithm to only inline one call site at a time. Unfortunately, this would require very long training times in order for the learner to be observe the effects of inlining all the sites.

The fourth problem seems to be the most likely cause of the poor performance, and it is also the easiest to explore further. The current algorithm performs a single gradient descent update step for each time the benchmark is inlined and run. This is a huge waste of the precious data collected from running the benchmark (precious because its collection represents the overwhelming majority of the time used by the system). A gradient descent learner does not use all the information present in the training data in just a single pass over it. The learner could easily be improved by iterating over the collected data many times. Because the data would only be collected once, this would not take much longer than the learner currently takes. Unfortunately, if the learner learned from data only after it was collected, it would not be able to influence the sets of call sites inlined during the collection of data. A second modification would therefore be for the learner to iterate over the collected data as it is being collected. For example, the learner could keep track of the running time of the last 100 inlinings, and before measuring the running time of the next inlining, it could make many training passes over this previously observed data.

References

[1] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM Sigplan Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO-00)*, volume 35.7 of *ACM SIGPLAN NOTICES*, pages 52–64, 2000.

[2] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[3] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on Lisp and Functional Programming*, pages 273–282, 1994.

[4] Jerome Miecznikowski. Personal communication, 2001.

[5] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco, 2000.

[6] Manuel Serrano. Inline expansion: *When* and *How.* In *Programming Languages, Implementation and Logic Programming (PLILP)*, volume 1292 of *LNCS*, pages 143–157. Springer, 1997.

[7] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, volume 1781 of *LNCS*, pages 18–34. Springer, 2000.